Everything in perspective

Tijmen Joppe Muller

January 14, 2004

Technical Report No. ICT TR ??.????

Author
Tijmen Joppe Muller
tijmen@avpec1910.nl

Supervisor
Jonathan Gratch
gratch@ict.usc.edu
Institute for Creative Technologies
13274, Fiji Way
Marina del Rey, California 90292-7008
United States of America

Mentor
Anton Nijholt
anijholt@cs.utwente.nl
University of Twente
Postbus 217
7500 AE Enschede
The Netherlands

For the outlining of this document LaTeX 2_ε has been used.
http://mullert.adsl.utwente.nl/

Acknowledgement
This work was funded by the Department of the Army under contract DAAD 19-99-D-0046. Any opinions, findings, and conclusions expressed in this article are those of the authors and do not necessarily reflect the views of the Department of the Army.

Preface

A compulsary part within the study of Computer Science at the University of Twente is a 14 week internship. Arno and I have chosen to perform this internship abroad, at the Institute for Creative Technologies. The internship consists of two assignments, both within the Mission Rehearsel Exercise project. The first assignment, concerning a dialogue system on emotions, is discussed in the report *Interaction on emotion*. This report, *Everything in perspective*, discusses the second assignment.

I'd like to thank Fast Eddie for giving me a lead on how to implement the tracker device, helping me debugging my code and giving me hints on coding in C++ in general. Thanks go to Jonathan Gratch for making the internship possible and for taking the time and effort to guide us. Finally, I want to thank Anton Nijholt for his guidance during the internship.

January 14, 2004, TJM

Contents

Chapter 1

Introduction

The virtual world of the Mission Rehearsal Exercise project is three dimensional, but the scenery is projected on a 2D screen. As the user moves around in the virtual theatre, the view on the scenery should change accordingly. The goal of this assignment is to gather experience on camera movement to make the projection on the screen realistic.

Chapter 2

Requirement specification

In figure 2.1 the situation is drawn schematic: if the user at location P steps to the right to location P', an object like the building with diagonal LR projected on the 150 degree screen has to change from $S_L S_R$ to $S'_L S'_R$. The first part of the assignment is to experiment if it is sufficient to move the virtual camera (the 'eyes' in the virtual world) around, identical to the user's movements.

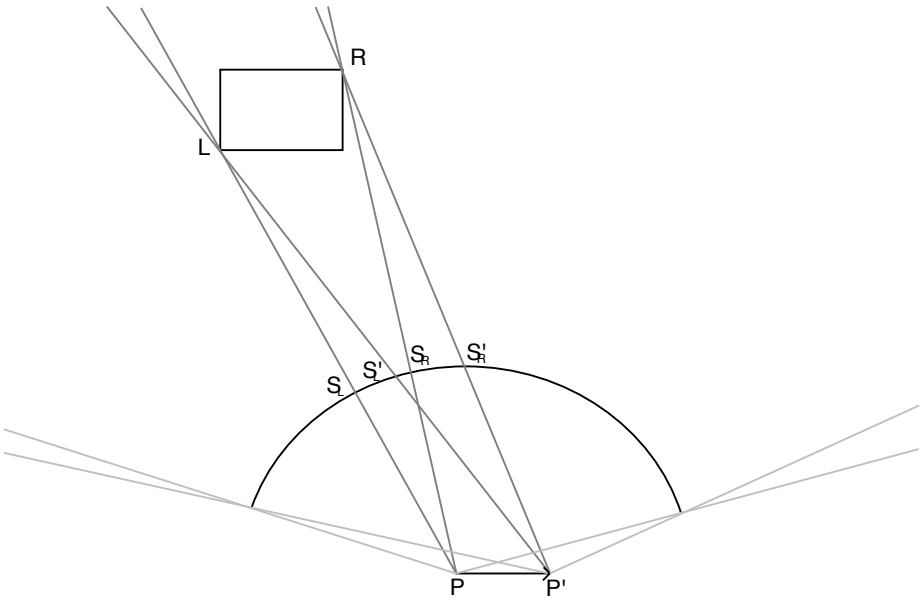

Figure 2.1: Projection diagram

Initially, the mouse and/or the keyboard can be used to provide data on the location of the user. Eventually, the head tracking device needs be implemented to provide the data, but this is not a requirement in the assignment.

The second part of the assignment is about the gaze of the agents. As the user moves around in the VR theatre, the agents that have their focus on the user should follow him.

Chapter 3

Analysis

3.1 Overview of the MRE system

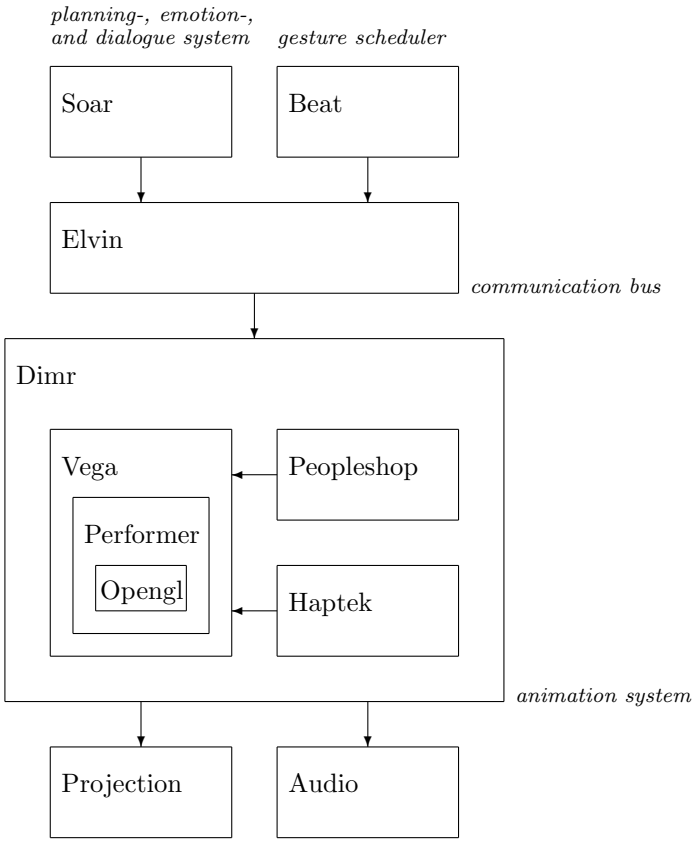

Figure 3.1: Overview of the MRE system

Figure 3.1 gives an partial overview of the MRE system. The geometry

of the environment (the static objects), the paths for animated objects and the camera's are defined in *Vega*, a software environment build on top of *Performer*, which subsequently is built on top of the core graphics engine *OpenGL*. For the animation of the characters *Peopleshop* is used, which subsequently uses *Haptek* for the animation of their faces. Peopleshop instructs Vega to draw the geometries of the animated characters.

Elvin acts as a communication bus between an outside user, for example the Soar system or the gesture scheduler Beat, and *Dimr*. The commands received by Dimr are parsed and translated into commands for Vega, Peopleshop or other parts of the system not shown.

For defining and previewing Vega applications the graphical user interface *LynX* can be used. It provides in the possibility to create the environment, but also paths and motion models. More on these subjects in section 3.2.

3.2 Camera motion

There are five approaches towards handling the camera:

A path. A *path* is defined as a collection of waypoints. After creating a path in LynX, an observer can traverse the path, applying the predetermined motion [VLH01]. Because of this predetermination, this option is not useful for the assignment, since the user of the system has total freedom of movement within a certain area.

An existing motion model. *Motion models* are positioning and motion methods that can be attached to an observer like the camera. By using an input device, each motion model type allows the user to interactively control the position and orientation of the attached observer. There are nine pre-defined motion model types, which can be adjusted by different parameters. For a detailed explanation of the motion models, see [VLH01], section 3.13.

One of the manipulations on a motion model is the use of an *isector*. An isector defines how the motion model behaves when it intersects with other parts of the landscape, such as the terrain. For example, it can position the motion model automatically at the intersected terrain elevation, so the user always stands 'on the ground'. [VLH01]

The question is whether one or a combination of the existing models is sufficient for the purposes of this assignment. This seems to be the case: the *Warp* model transforms location of the observer using the data provided by the input device as x- and y-coordinates on a part of the scenery's map. This area can be defined with the model's parameters, so a one-to-one mapping of this area with the real area in the theatre should be sufficient for this assignment.

The disadvantage of this method is that the location of the observer is defined by the location of the input device on top of the output screen of Vega. For example, if the mouse is used as the input-device, the movement of the mouse over the output screen causes the movement of the observer in the defined area. As the size of the output screen can easily differ in various situations, it is hard to make this method reliable.

A new motion model. It is possible create and define up to six additional motion models into the Vega kernel at one time. This would require a lot of research of the Vega architecture, since the specialist on this area is not available until late January. [VFH??]

Manual commands. Instead of directly using Vega, commands can be send to Dimr, which translates it into commands for Vega. To manually place the camera at a certain position with a certain angle, the following commmand is used:

```
dimr vega observer <name> pos <x> <y> <z> <h> <p> <r>
```

The meaning of the arguments is the following: `<name>` the name of an observer, i.e. 'camera', `<x> <y> <z>` viewing location coordinates, `<h> <p> <r>` head, pitch and roll, the viewing angle.

An application that polls the input device constantly and sends the correct Dimr commands should be convincing enough for realistic movement of the camera.

Integration in Dimr. The last option is, instead of creating a stand-alone application, integrating the polling library into Dimr. This would require Dimr to be recompiled, though. It means the code for perspective cannot be changed easily, nor can it be easily deactivated.

3.3 Tracker device

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
x	−2.16	−0.40	1.20	−1.15	0.62	2.35	−0.25	1.48	3.15	2.18
y	−1.25	−0.42	1.86	−2.30	−0.6	1.10	−3.15	−1.46	0.25	2.19

Table 3.1: Results for ten positions

The tracker used in the VR theatre is an InterSense IS-900 [IS900], which consists of a number of strips constructed to the ceiling and up to 4 tracking devices; these can be a helm, a pistol-like tracker, et cetera. It is posible to track the x-, y- and z-coordinates (relatively to an origin, in meters) and heading, pitch and roll angles, but in this experiment only the x- and y-coordinates will be used.

To find out in what way the VR theatre is projected onto the IS-900, a number of measurements were done. The area of the VR theatre the user can move around in has more or less the shape of a house with a rounded roof[1]. The projection screen is located around this 'roof'. The found coordinates by the IS-900 for ten positions on the area are listed in table 3.1, point A being the left most positions away from the screen and I being the right most position near the screen. Shown in scaled diagram 3.2, it's clear that the InterSense model rotates and flips the real area.

[1] Please, use your imagination.

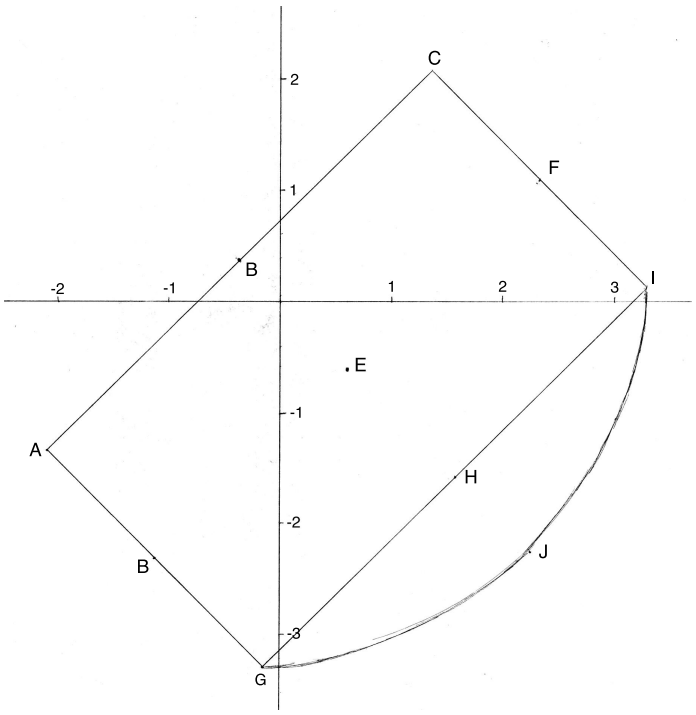

Figure 3.2: Scaled diagram of the VR theatre

3.4 Conclusion

Considering all options, creating a stand-alone application which sends commands to Dimr directly seems to be the best. The application will need to transform the model by the tracker to the real area. The device will be implemented immediately (so no keyboard/mouse-input), using a polling library supported by the Intersense company [INT03]. Additionally, the generality of such a application promises long-time usability.

Chapter 4

Mathematical model

4.1 Problem definition

The projection of the scenery needs to depend on the position of the user in the VR theatre. The tracker device provides this data, but it has to be translated to a position for the camera in the virtual world, since the areas don't map on top of each other precisely. This is due to the construction and configuration of the IS-900 in the theatre. The purpose of this chapter is to give a model that can be used to create an application for this experiment.

4.2 Formalization

4.2.1 Assumptions

- The unit for distance for both the tracker device and the virtual world are meters.
- The starting point (or *origin*) of the user in the virtual world is variable: $O(x_O, y_O)$.

4.2.2 Translation

To correctly map the user's position $P(x, y)$ in the real world to the virtual camera's position $P_{virtual}(x_{virtual}, y_{virtual}$, a rotation and a flip is necessary. If the rotation is executed before the flip, only a simple horizontal flip is needed (y-axis as mirror). Finally, the origin of the real area must be placed over the starting point of the user in the virtual world.

For the rotation over θ degrees of point p' the following formula is used:

$$\mathbf{p}' = \mathbf{R}\mathbf{p} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{4.1}$$

The horizontal flip:

$$\mathbf{p}'' = \mathbf{h}\mathbf{p}' = \begin{bmatrix} -1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{4.2}$$

Finally, after placing the virtual camera at the starting point, $P_{virtual}$ is acquired:

$$\begin{bmatrix} x_{virtual} \\ y_{virtual} \end{bmatrix} = \mathbf{p} + \mathbf{t} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x_O \\ y_O \end{bmatrix} \tag{4.3}$$

4.2.3 Analysis

1. First step is the rotation (equation 4.1):

$$\mathbf{p}' = \mathbf{R}\mathbf{p} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix} \tag{4.4}$$

2. Substitute 4.4 into 4.2:

$$\mathbf{p}'' = \mathbf{h}\mathbf{p}' = \begin{bmatrix} -1 & 1 \end{bmatrix} \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix} = \begin{bmatrix} -x\cos\theta + y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix} \tag{4.5}$$

3. Substitute 4.5 into 4.3:

$$\begin{bmatrix} x_{virtual} \\ y_{virtual} \end{bmatrix} = \mathbf{p}'' + \mathbf{t} = \begin{bmatrix} -x\cos\theta + y\sin\theta + x_O \\ x\sin\theta + y\cos\theta + y_O \end{bmatrix} \tag{4.6}$$

4.3 Results

The numbers found in section 3.3 assume the rotation angle θ is 135 °. The final formulas for $x_{virtual}$ and $y_{virtual}$ then are:

$$x_{virtual} = \frac{1}{2}\sqrt{2}x + \frac{1}{2}\sqrt{2}y + x_O$$

$$y_{virtual} = \frac{1}{2}\sqrt{2}x - \frac{1}{2}\sqrt{2}y + y_O$$

The coordinates of origin O are left as arguments for the application, since it is very possible these numbers differ in different scenarios. The variables x and y are of course input from the tracker device.

Chapter 5

Design and implementation

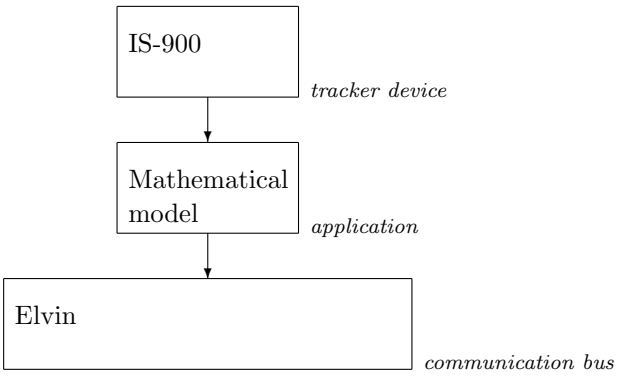

Figure 5.1: Dataflow

The dataflow for the application is very straightforward. The objective is to provide Dimr with the information needed to change the camera stance. This is done by sending commands to the communication bus Elvin. Input is taken from the tracker device and this data is processed as described by the mathematical model in chapter 4. This model, depicted in figure 5.1, can be seen as an addition to the model in figure 3.1 (page 5).

The application takes up to three arguments, that is a transformation for the x-coordinate, a transformation for the y-coordinate and a rotation. Last mentioned is added because it's not sure if this factor is constant and it was not hard to make an argument out of this. For handling of the positions the datatype *float* is used, and the data is rounded up to millimeters. The tracker device is polled 24 times per second (this number is chosen because it equals the 'frame rate' of the human eye) and commands are sent to Dimr at the same rate (if the position has changed, naturally).

There were no problems[1] during the implementation. The source code of the final version can be found in appendix A.

[1] Actually, there were a million problems during implementation, but they were all due to a lack of C++ experience of the author. Which, of course, is not worth mentioning.

Chapter 6

Testing

Testing was done by actually running the application in the MRE scene in the VR theatre in the presence of several people. This resulted in the following observations:

- Despite of the frequent refreshes of the position and the high frame rate of the projection (around 30), the movement seems to take place in steps, which makes the movement jumpy.
- When closer to the screen, the objects in the virtual world seem too big. Also, the movement towards the screen doesn't seem to be in the right proportions; i.e. if the user moves one step towards the screen in the real world, the actual distance travelled seems to be more than one step.
- The agents are following the user with their eyes and body movements, but not completely convincing to some.
- The application does not print out all the information written to the command line (like the detection of the Intersense device, the settings for the translation and rotation).

These problems are the starting point for a second version, of which the results are descibed in the next chapter.

Chapter 7

Iteration

This chapter describes the changes that have been made after testing the first version of the application.

7.1 Fluent motion

In the first version the motion of the camera was not as fluent as it should be, as noticed during testing. After debugging Dimr, it became clear the messages it receives from the first version application for changing the camera stance is rounded to centimeters instead of millimeters. Some external possibilities for this to happen have been researched, e.g. the use of the *float* type, the frame rate of the projection by Vega and the commands from Dimr to Vega, but these are not the cause. The problem is the code that converts the values provided by the mathematical model from floats to a string: it discards a digit. After improving this code, the movement improved significantly.

7.2 Missing information

The fact that not all information was correctly written to the command line was caused by a missing C++-command: output needs to be *flushed* to be certain it is presented to the user.

7.3 Scaling

In order to experiment with different motion models, a scaling factor has been added to the application. Each translated coordinate is multiplied by the scaling factor s:

$$x_{virtual} = (\frac{1}{2}\sqrt{2}x + \frac{1}{2}\sqrt{2}y)s_(x) + x_O$$

$$y_{virtual} = (\frac{1}{2}\sqrt{2}x - \frac{1}{2}\sqrt{2}y)s_(y) + y_O$$

7.4 Vertical movement

The tracker device supports not only tracking horizontal movement, but vertical movement as well. The command to Dimr to change the camera stance accepts a z-coordinate, so after defining the mathematical model for the z-coordinate vertical movement had been implemented as well (i.e. if the user jumps or crouches in the real world).

The mathematical model for the z-coordinate is a breeze in comparison to the x- and y-coordinates. The z-coordinate provided by the tracker device is flipped in the same way as the x-coordinate is. After including a scaling factor s_z and a transformation z_O, the following formula applies:

$$z_{virtual} = -zs_z + z_O$$

Since the tracker used in testing is held in hand instead of placed on the head, it still must be possible to set the z-coordinate to a fixed value instead of reading the position from the tracker device; this is done by a command line argument (see section 7.5).

7.5 Command line arguments

The first version accepts three arguments on startup: a x- and y-transformation and a rotation. This syntax is inflexible and there were some desired extensions, so this has been changed considerably in the second version.

The various arguments accepted by the second version are best presented by it's own help:

```
-h                  display this help and exit
-r <degr>           rotate by <degr> degrees
-s <xf> <yf> <zf>   scaling factors
-t <x> <y> <z>      transfrom origin by <x>, <y> and <z>
-u <rate>           set update rate to <rate>
-v                  display debug information
-z <height>         set z-coordinate to fixed value <height>
```

To start the application specifically for the MRE Bosnia scenario and using the hand tracker device, the following line should be entered at the command prompt:

```
./persp -r 135 -t -10.7 -34.7 0 -z 2
```

Note It is important that the environment variable `ELVISH_SCOPE` is set to the right value, otherwise Dimr won't receive the messages generated by the perspective application!

Chapter 8

Conclusion and recommendations

The resulting application of this assignment gives a good first[1] implementation of the tracker device. The mathematical model is correct for the current set up of the VR theatre at the Institute of Creative Technologies and the movement on the projection is smooth and convincing to a high degree. The application as is provides a good starting point for further research on incapsulating the tracker into the MRE project.

Some points that deserve attention in further research are:

Floating The vertical ('eyeheight') position in the virtual world does not depend on the 'height' of the ground – the camera is actually floating in the world, instead of standing on the ground. As a consequence, if the landscape has differences in height, the camera won't adjust to those differences. This is hardly noticed in such a small area as in the VR theatre, but implementing this would increase realism.

Frustrum It is very possible the current motion model of the camera moving around in the virtual world is not correct. As with the former item, it is hard to notice in the current implementation, but if one stands really close to the screen, the virtual agents are too big. A theory is that the projection frustrum needs to be changed – some research needs to be done here.

[1] And second actually, since there already is a second version.

Appendix A

Source code

A.1 perspective.h

```
#ifndef __DIMR_APP_H
#define __DIMR_APP_H

#define DIMR_OK 0
#define DIMR_ERR 1

#define MAX_CMD_ARGS 64
#define MAX_CMD_ARGL 1024

#endif // __DIMR_APP_H

#define PI              3.14159265
```

A.2 perspective.cxx

```
#include <math.h>
#include <time.h>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include "perspective.h"

#include "../intersense/intersense.h"
#include "tt_utils.h"

IBOOL                   verbose;
ISD_TRACKER_HANDLE      handle1;
ISD_TRACKER_TYPE        Tracker1;
ISD_STATION_CONFIG_TYPE Station1[ISD_MAX_STATIONS];
ISD_DATA_TYPE           data1;
WORD                    station;
```

```
float                   degr, fixedz;
float                   scalex, scaley, scalez;
float                   transfx, transfy, transfz;
float                   xpos, ypos, zpos;
int                     debug, err, updaterate, same, zIsFixed;

int main(int argc, char **argv)
{
   // Print out information
   cout << "\nTracker device for MRE, version 1.01\n";
   cout << "by Tijmen Joppe Muller, 01/14/2004\n";
   cout << "tijmen@avpec1910.nl\n";
   cout << "http://mullert.adsl.utwente.nl/\n\n";

   // Default values
   debug = 0;
   degr = 0;
   updaterate = 24;
   scalex = 1;
   scaley = 1;
   scalez = 1;
   transfx = 0;
   transfy = 0;
   transfz = 0;
   zIsFixed = 0;

   // Process arguments
   int i = 1;
   while (i < argc)
   {
      // Set update rate
      if (!strcmp(argv[i], "-u"))
      {
         if (argc - i <= 1)
         {
            cout << "Wrong argument: no value for update rate.\n";
            return -1;
         }

         // Get next argument (update rate)
         i++;

         int tempupdaterate = atoi(argv[i]);
         if (tempupdaterate < 1)
         {
            cout << "Wrong argument: update rate needs ";
            cout << "to be an integer greater than 0.\n";
            return -1;
         }
         else updaterate = tempupdaterate;
      }

      // Print out help info
      else if (!strcmp(argv[i], "-h"))
```

```
{
   cout << "Usage: ./persp [options]\n";
   cout << "Options:\n";
   cout << "   -h                  ";
   cout << "display this help and exit\n";
   cout << "   -r <degr>           ";
   cout << "rotate by <degr> degrees";
   cout << ", default [0]\n";
   cout << "   -s <xf> <yf> <zf>   ";
   cout << "scaling factors,";
   cout << " default [1 1 1]\n";
   cout << "   -t <x> <y> <z>      ";
   cout << "transfrom origin by <x>, <y> and <z>,";
   cout << " default [0 0 0]\n";
   cout << "   -u <rate>           ";
   cout << "set update rate to <rate>";
   cout << ", default [24]\n";
   cout << "   -v                  ";
   cout << "display debug information\n";
   cout << "   -z <height>         ";
   cout << "set z-coordinate to fixed value <height>\n";
   return 0;
}

// Set rotation
else if (!strcmp(argv[i], "-r"))
{
   if (argc - i <= 1)
   {
      cout << "Wrong argument: no value for rotation.\n";
      return -1;
   }

   // Get next argument (actual rotation)
   i++;

   float tempdegr = atof(argv[i]);
   if (tempdegr < 0 || tempdegr >= 360)
   {
      cout << "Wrong argument: rotation needs to be";
      cout << " between 0 and 360 degrees.\n";
      return -1;
   }
   else degr = tempdegr;
}

// Set scaling
else if (!strcmp(argv[i], "-s"))
{
   if (argc - i <= 3)
   {
      cout << "Wrong argument: no value(s) for x-, y-";
      cout << " and/or z-scaling factor.\n";
      return -1;
```

```
   }

   // Get next argument (x-transformation)
   i++;
   scalex = atof(argv[i]);

   // Get next argument (y-transformation)
   i++;
   scaley = atof(argv[i]);

   // Get next argument (z-transformation)
   i++;
   scalez = atof(argv[i]);
}

// Set transformation
else if (!strcmp(argv[i], "-t"))
{
   if (argc - i <= 3)
   {
      cout << "Wrong argument: no value(s) for x-, y-";
      cout << " and/or z-transformation.\n";
      return -1;
   }

   // Get next argument (x-transformation)
   i++;
   transfx = atof(argv[i]);

   // Get next argument (y-transformation)
   i++;
   transfy = atof(argv[i]);

   // Get next argument (z-transformation)
   i++;
   transfz = atof(argv[i]);
}

// Verbose mode
else if (!strcmp(argv[i], "-v"))
{
   debug = 1;
}

// Fix z-axis
else if (!strcmp(argv[i], "-z"))
{
   if (argc - i <= 1)
   {
      cout << "Wrong argument: no value";
      cout << " for fixed height.\n";
      return -1;
   }
```

```
         // Get next argument (actual rotation)
         i++;

         fixedz = atof(argv[i]);
         zIsFixed = 1;
      }

      // Argument not recognized, so ignored
      else
      {
         cout << "Unknown argument \"";
         cout << argv[i] << "\" ignored.\n";
      }

      // Jump to next argument
      i++;
   }

   cout << "Update rate set to ";
   cout << updaterate << " messages per second.\n";
   cout << "Rotation set to ";
   cout << degr << " degrees.\n";
   cout << "Scaling set to ";
   cout << scalex * 100 << "% for x, ";
   cout << scaley * 100 << "% for y, ";
   cout << scalez * 100 << "% for z.\n";
   cout << "Transformation set to ";
   cout << transfx << " on x-axis, ";
   cout << transfy << " on y-axis, ";
   cout << transfz << " on z-axis.\n";
   if (zIsFixed)
   {
      cout << "Height set to fixed value ";
      cout << fixedz << ".\n";
   }
   if (debug) cout << "Verbose mode.\n";

   // Connect with elvin
   cout << "\nOpening connection to Elvin... ";

   err = ttu_open("caldera.ict.usc.edu");
   if (err == TTU_SUCCESS)
   {
      cout << "done.\n";

      // Registering for receiving messages.
      // Not really necessary, since no messages
      // are ever received.
      ttu_register( "all" );
      ttu_register( "vrAllCall");
      ttu_register( "vrStop");
   }
   else
   {
```

```
   // Exit on fail
   cout << "failed.\n";
   return -1;
}

// Initialize the tracker device
cout << "Opening connection to tracker device...\n";
cout.flush();
handle1 = ISD_OpenTracker(0, FALSE, verbose);

// Exit on fail
if (handle1 == -1)
{
   cout << "Connection to tracker device failed.\n";
   ttu_close();
   return -1;
}

// Get tracker configuration info
cout << "\nConfiguring tracker.\n";

ISD_GetTrackerState(handle1, &Tracker1, verbose);

cout << "Checking model... ";

switch (Tracker1.TrackerModel)
{
   case ISD_UNKNOWN:
      cout << "failed, unknown.\n";
      err = TTU_ERROR;
   case ISD_IS300:
      cout << "failed, IS-300.\n";
      err = TTU_ERROR;
   case ISD_IS600:
      cout << "failed, IS-600.\n";
      err = TTU_ERROR;
   case ISD_IS900:
      cout << "done, IS-900.\n";

      for (station = 1; station <= 4; station++)
      {
         // fill ISD_STATION_CONFIG_TYPE structure
         // with current station configuration
         if (!ISD_GetStationState(handle1,
              &Station1[station-1], station, verbose))
         {
            cout << "Get station ";
            cout << station << " state failed.\n";
            err = TTU_ERROR;
            break;
         }

         // change flags from default FALSE state
         Station1[station-1].GetButtons = TRUE;
```

```
            Station1[station-1].GetAnalogData = TRUE;
            Station1[station-1].AngleFormat = ISD_EULER;

            // apply new configuration
            if (!ISD_SetStationState(handle1,
                 &Station1[station-1], station, verbose))
            {
               cout << "Set station ";
               cout << station << " state failed.\n";
               err = TTU_ERROR;
               break;
            }
         }

         break;
      case ISD_INTERTRAX:
         cout << "failed, InterTrax.\n";
         err = TTU_ERROR;
   }

   // Exit on fail
   if (err == TTU_ERROR)
   {
      ttu_close();
      return -1;
   }

   cout << "Checking headtracker for changes.\n";

   // Flush
   cout.flush();

   while (1)
   {
      // Get data from the tracker device
      ISD_GetTrackerData (handle1, &data1);

      // Which station?
      int tracker = 0;

      // For x- and y-coordinates, get coordinates from
      // tracker, calculate values, scale them and round
      // results to millimeters
      float trackx = data1.Station[tracker].Position[0];
      float tracky = data1.Station[tracker].Position[1];

      float calcx = - cos(degr * PI / 180) * trackx;
      calcx += sin(degr * PI / 180) * tracky
      calcx = calcx * scalex + transfx;
      float calcy = sin(degr * PI / 180) * trackx;
      calcy += cos(degr * PI / 180) * tracky;
      calcy = calcy * scaley + transfy;

      float newxpos = ceil(calcx *  1000) / 1000;
```

```
      float newypos = ceil(calcy *  1000) / 1000;

      // For z-coordinate, if the height is not fixed,
      // get tracker data, calculate, scale and round
      // otherwise position on z-axis is always the same
      float newzpos = fixedz;
      if (!zIsFixed)
      {
         float trackz = data1.Station[tracker].Position[2];
         float calcz = -trackz * scalez + transfz;
         newzpos = ceil(calcz * 1000) / 1000;
      }

      // Compare tracker data with current position
      // If data changed, update camera stance accordingly
      if (xpos != newxpos || ypos != newypos || zpos != newzpos)
      {
         if (debug)
         {
            cout << "Changed position (" << xpos << ", ";
            cout << ypos << ", " << zpos << ") ";
            cout << "to (" << newxpos << ", ";
            cout << newypos << ", " << newzpos << ")\n";
         }

         // Update coordinates
         xpos = newxpos;
         ypos = newypos;
         zpos = newzpos;

         // Send elvin message to change camera position
         char camera[100];
         sprintf(camera, "vega observer camera pos
                          %f %f %f 5 -5 0", xpos, ypos, zpos);

         // Send message
         ttu_notify2("dimr", camera);

         // Set boolean that position has changed
         same = 0;
      }
      else if (debug)
      {
         if (!same) cout << "Position stayed the same...\n";
         same = 1;
      }

      // Doortrekken
      if (debug) cout.flush();

      // Wait until next update
      sleep(1/updaterate);
   }
```

```
   // close the device
   ISD_CloseTracker(handle1);

   // close the lib
   ttu_close();

   return 0;
}
```

Bibliography

[VLH01] MultiGen-Paradigm Inc.: Vega Lynx User's Guide (March 2001)

[VFH??] MultiGen-Paradigm Inc.: Vega Function Help

[IS900] InterSense: Technical Overview IS-900 Motion Tracking System - `http://www.isense.com/support/downloads/IS900_Tech_Overview_Enhanced.pdf`

[INT03] InterSense, Inc. - `http://www.isense.com/`