Technical Report ECE.P54.2006.7
December 4, 2006

Bird's-Eye Tracking Application for the DriveSafety Driving Simulator

Oskar Palinko

ECE Department, University of New Hampshire

TABLE OF CONTENTS

1. INTRODUCTION 3

2. USER MANUAL 3

3. TECHNICAL DESCRIPTION 6

3.1 SIMULATION 6
3.2 WINSOCKET SERVER 7
3.3 GRAPHICAL INTERFACE 8
3.4 CONFIGURATION FILE 11

1. Introduction

Version 1.9.35a of the DriveSafety simulator software does not provide an ability to track the entities in the simulation from a bird's-eye view during the execution of experiments. This gap is filled by the application described in this report.

The first part, the user manual, is intended to give step-by-step instructions on how to use this software with an arbitrary simulator scenario.

The second part is intended to give technical information on the structure of the application.

2. User manual

There are a few preparatory steps which must be followed to have the application work properly with an arbitrary simulator scenario project.

1. A snapshot of the scenario must be taken. To do this, start HyperDrive and load the desired project. Maximize the map size by maximizing the HyperDrive window and by zooming in.

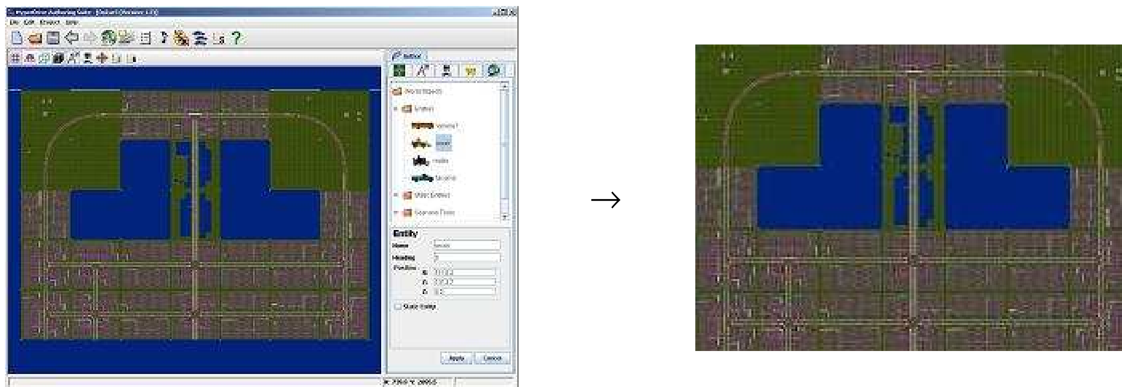

Figure 2.1 Cropping the appropriate part of the image

Take a snapshot by pressing the print-screen key on your keyboard. Start an image processing program, e.g. Paint. Paste the snapshot from the clipboard. Crop the image to the extent of the map, as show in Figure 2.1. Save the image under an arbitrary name. The image type can be either BMP or JPG.

2. Consider the image: calculate its width (scene_x) and height (scene_y) in meters using HyperDrive. The distance between two adjacent white dots in the map grid in HyperDrive amounts to 200m.

3. Calculate the offset of the map from the left bottom corner of the scenario.

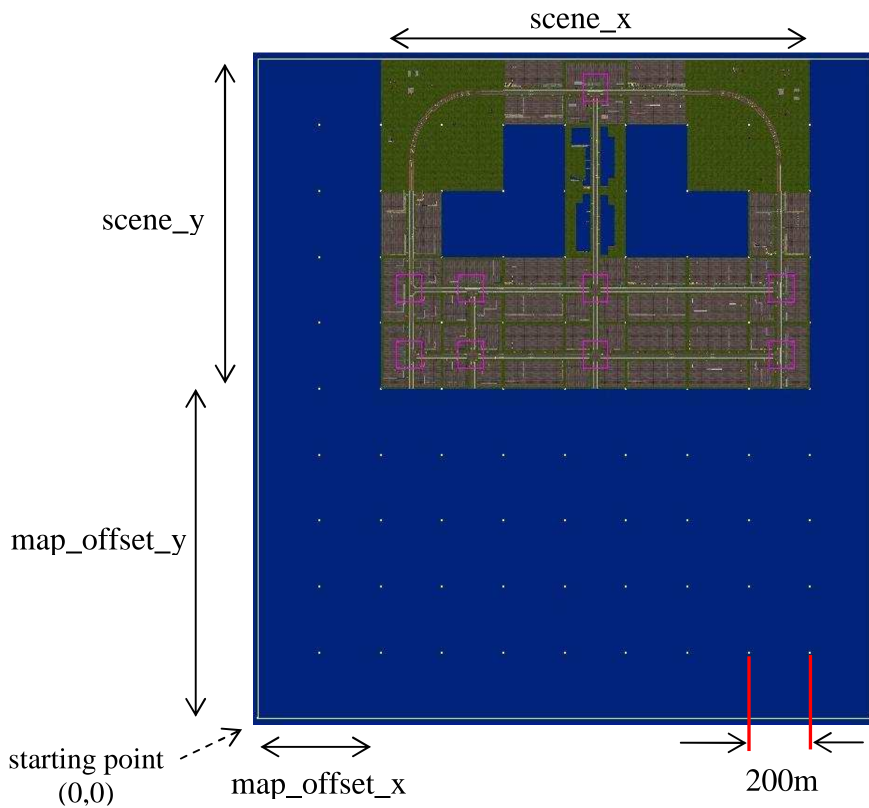

Figure 2.2 Important distances in the simulation scenario

In the particular case of the above Figure 2.2 the variables have the following values:

scene_x = 1400
scene_y = 1000
map_offset_x = 400
map_offset_y = 1000

4. Find the config.txt file in the Tracking Application's folder. Edit it. In the first line enter the path and name of the snapshot file, using C-style double backslashes, e.g.:

C:\\My Folder\\track_map.bmp

In the next lines enter the acquired values for scene_x, scene_y, map_offset_x and map_offset_y. Save the file. It should look like this:

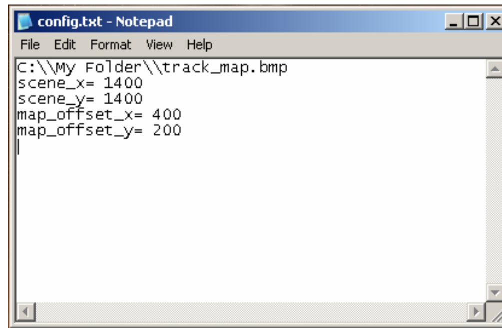

Figure 2.3 config.txt

5. Add the following snippet of code to the initialization script of your HyperDrive project:

```
set iii 0
set ::hostIP 192.168.10.159
set ::hostPort 4242
set ::ndSocket [socket 192.168.10.159 4242]
fconfigure $::ndSocket -buffering none -blocking false
set numEntities [SimGetNumEntities]
set veh_numb 10

if {$numEntities < $veh_numb} {
    set veh_numb [expr $numEntities + 1]
}

VTriggerCreate vtSecond {
    VisualsDisplayText textsend2 1 0.2 0.9 1 0 200 0 PERMANENT  $Time

    set iii [expr $iii + 1]
    set uuu [expr $iii % $veh_numb]
    set closest [EntityGetClosestEntities Subject $veh_numb Vehicles]
    if {$uuu==[expr $veh_numb -1]} {
        set tex [format "subject  %d %d %d " [expr round($SubjectX)]  [expr
round($SubjectY)] [expr round($SubjectHeading)] ]
        VisualsDisplayText textf 1 0.2 0.7 1 0 200 0 PERMANENT $tex
        puts $::ndSocket  $tex
    } else {
        set Ent [lindex $closest [expr $uuu*3]]
        set EntAr [SimOutputEntityArray $Ent]
        set EntX [lindex $EntAr $::EntityXIndex]
        set EntY [lindex $EntAr $::EntityYIndex]
        set EntHead [lindex $EntAr $::EntityHeadingIndex]
        set tex [format "%s %d %d %d " [lindex $closest [expr $uuu*3]] [expr
round($EntX)] [expr round($EntY)] [expr round($EntHead)]]
        VisualsDisplayText textf 1 0.2 0.7 1 0 200 0 PERMANENT $tex
        puts $::ndSocket  $tex
    }
}
VTriggerAdd vtSecond 10 Hz
```

This code first initializes the communication channel towards the host application that is located on a computer with the IP address 192.168.10.159. Then a trigger is created that will be executed 10 times per second. The number of displayed vehicles is defined by the variable `veh_numb`. Save and upload the project, but do not start it yet.

6. Run the Tracking Application, then start the scenario from Dashboard.

3. Technical description

The simulator and the application are connected via a TCP/IP network connection. The running scenario must include the TCL/TK code snippet which allows the sending of required data to the tracker software. The communication is one-way, there is no feedback information sent from the application to the simulator.

The above mentioned TCL/TK code of the scenario extracts a number of vehicle entities (ambient traffic included) in the vicinity of the subject and sends their data to the tracker. These include: entity name, x-position, y-position and heading.

The application is written in Visual C++ as an MFC project. It utilizes a graphical environment instead of the usual form-controls-type windows. No additional graphics engine was used (e.g. DirectX, OpenGL) because MFC's native functions proved to give satisfactory results.

Some of the important parts of the project can be visualized in the following way:

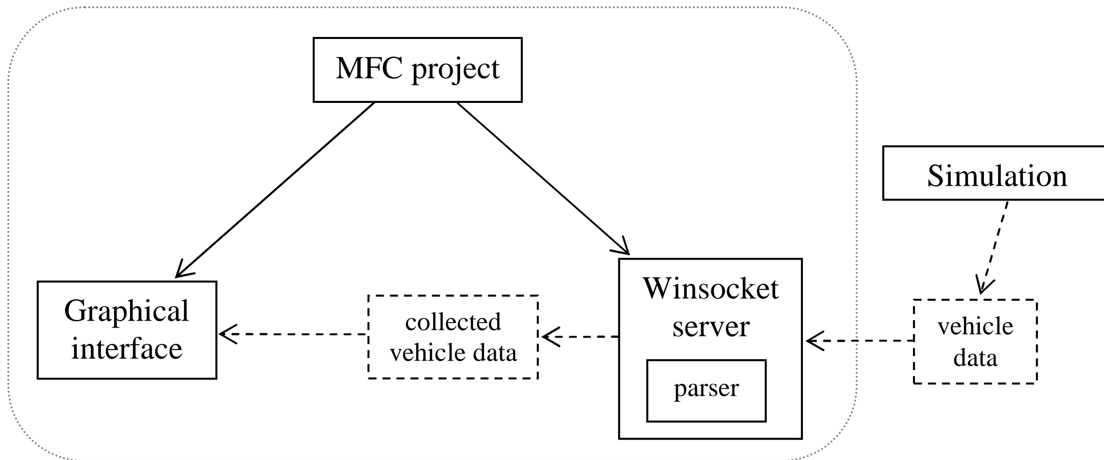

Figure 3.1 Elements of the project and simulation

The flow of information starts at the simulation.

3.1 Simulation

The TCL/TK code listing in chapter 2 is responsible for sending the appropriate data to the tracking application. The `vtSecond` virtual trigger function executes 10 times in a second. It finds `veh_numb` number of vehicles closest to the subject. It collect their

data and sends them over the TCP/IP network to the Winsocket server of the application. Each time it sends a sequence of the following message:

Vehiclename positionx positiony heading

For example, a sequence could be:

Subject 1402 1023 90
Ambient125 1234 932 180
Yellow_golf 1503 1339 85

Vehiclename is the name of the entity in the simulation. Positionx and positiony are the x and y distances, given in meters, calculated from the bottom-left starting point of the simulation scenario (see Figure 2.2). Heading is the heading of the vehicle in the simulation in degrees (0 degrees means north, 180 degrees south).

3.2 Winsocket server

The Winsocket server is implemented as a separate thread. The server-thread is started from the OnInitialUpdate method of the CBirdseyeView class by calling the PowerUpServer() function. The server's elements are defined in the "basic-server.cpp" file. It accepts connections from the simulator. It means, that the tracking application must be started before starting the simulation, because the server must be ready for the connecting client.

Each time the tracker receives a message of the mentioned form, it parses the string, separating the vehicle_name, x & y positions and the direction into the columns of a 2 dimensional array, namely `stc`. The first row in this table is always the data collected from the subject (the row counter resets each time the word "subject" is received). Then the other entities follow:

subject	1402	1023	90
Ambient125	1234	932	180
Yellow_golf	1503	1339	85
…			

Table 3.1 The layout of the `stc` 2D array variable

The maximum number of rows in this variable is limited indirectly by `veh_numb`. If there are less vehicles in the simulation, then of course the no. of rows is less than `veh_numb`.

3.3 Graphical interface

The graphical interface has different tasks. First it loads and displays the background image, which represents the map of the scenario. The way to create this image is explained in chapter 2. The manipulation (loading, scrolling) is done using a custom class, CPicture, which relies on the standard IPicture interface. CPicture is declared and defined in “Picture.h” and “Picture.cpp”. Only two of its methods are called from the application:

BOOL CPicture::Load(LPCTSTR pszPathName)

BOOL CPicture::Render(CDC* pDC, CRect rc, LPCRECT prcMFBounds)

The first one is called to load the image, while the second one is called to display the image at an arbitrary position on the screen. “CRect cr” is used to define the rectangle for the rendering.

The graphical interface gets its data from the `stc` variable. The heading data is converted to radians, and the position data is converted from meters (of the simulation) to pixels (of the graphical display). The conversion formulas are derived as follows:

$$subj_x = (xcoord - map_offset_x)\frac{img_x}{scene_x}$$

$$subj_y = (scene_y - (ycoord - map_offset_y))\frac{img_y}{scene_y}$$

where

subj_x – screen coordinate in pixels,

xcoord – simulation coordinate in meters (starting form the bottom-left corner of the scenario),

map_offset_x – x distance between the bottom-left corner of the map and the bottom left corner of the scenario in meters,

img_x – the size of the map/image in pixels,

scene_x – the size of the map in meters.

The values for y are identically reasoned. The only difference is that ycoord-map_offset_y is subtracted from scene_y, because the y-coordinate in the simulation begins in the bottom (south) and its value rises upwards, while the zero of y in the graphical interface is in the top and its value is rising downwards. These measures are shown in Figure 2.2.

Most of the functionality of the graphical interface is done in the method OnDraw(CDC* dc) of the CBirdseyeView class. This is the standard “view” class of an MFC application. In the following, this method is explained, part by part.

```
      if(m_pict.Load(file_name))
      {
            CRect rcImage(CPoint(center_x - dynablock_x,center_y -
dynablock_y),m_pict.GetImageSize());
            CRect rc;
            m_pict.Render(pDC, rcImage);        // rendering the view
            CSize u = m_pict.GetImageSize(pDC);
            img_x = float(u.cx);                // getting the image
            img_y = float(u.cy);                // size
      }
```

The above part of the code loads the background image and displays it in the rc_Image rectangle. The actual position of the image depends on the center of display and the variables dynablock_x, dynablock_y, which are responsible for scrolling the image. Scrolling is done in a way, that when the arrow-image of the subject vehicle moves further than a 100 pixels away from the center of the screen (center_x,center_y) then the subject is re-centered together with the appropriate movement of the map.

In the next part, the main loop is started:

```
for (int ii=0; ii<=stc_length; ii++)
{
      if(ii==0)
      {
            xcoord = atoi(stc[ii][1]);    // extract x position
            ycoord = atoi(stc[ii][2]);    // extract y position
            int subj_x = int((xcoord - map_offset_x)* (img_x/scene_x))
            int subj_y = int((scene_y - (ycoord - map_offset_y)) *
(img_y/scene_y));       // conversion of x and y
            alpha = atoi(stc[ii][3]);     // extact heading
            alpha = 180 - alpha;
            double arad = (alpha*2*3.141592)/360;     // conv. angle
            int a = int(12*sin(arad));    //end point of the arrow
            int b = int(12*cos(arad));
            int mod_x;
            int delta_x;
            int mod_y;
            int delta_y;

            delta_x = subj_x-dynablock_x; // code for scrolling
            if (abs(delta_x)>100)         // if movement more that 100
            {
                  dynablock_x = subj_x;   // set dynablock for next
                  delta_x = 0;            // rendering of the image
            }
            mod_x = center_x + delta_x;   // mod for displaying arrow
            delta_y = subj_y-dynablock_y;
            if (abs(delta_y)>100)
            {
                  dynablock_y = subj_y;
                  delta_y = 0;
            }
            mod_y = center_y + delta_y;
      }
```

The main loop executes for every row of the `stc` array, i.e. for every vehicle. The first condition `if(ii==0)` is executed for the subject, the second one (not displayed in this snippet) for all others. Delta_x is the variable that measures how far the subject is from the center. If he is more than 100 pixels far, he is reset to the center. In the same time the map is scrolled by the same amount, so that the car stays in the same position compared to the map.

The next part of the code is responsible for displaying an arrow sign for the entities.

```
      CPoint lpPoint2;              // display the middle of the arrow
      lpPoint2 = CPoint(mod_x-a, mod_y-b);
      pDC->DPtoLP(&lpPoint2);
      pDC->MoveTo(lpPoint2);
      lpPoint2 = CPoint(mod_x+a, mod_y+b);
      pDC->DPtoLP(&lpPoint2);
      pDC->LineTo(lpPoint2);

      CPoint lpPoint3;              // display the right part of arrow
      lpPoint3 = CPoint(mod_x+a, mod_y+b);
      pDC->DPtoLP(&lpPoint3);
      pDC->MoveTo(lpPoint3);
      lpPoint3=CPoint(mod_x+int(8*sin(arad+0.8)),mod_y+int(8*cos(arad+
0.8)));
      pDC->DPtoLP(&lpPoint3);
      pDC->LineTo(lpPoint3);

      CPoint lpPoint4;              // display the left part of arrow
      lpPoint4 = CPoint(mod_x+a, mod_y+b);
      pDC->DPtoLP(&lpPoint4);
      pDC->MoveTo(lpPoint4);
      lpPoint4=CPoint(mod_x+int(8*sin(arad-0.8)),mod_y+int(8*cos(arad-
0.8)));
      pDC->DPtoLP(&lpPoint4);
      pDC->LineTo(lpPoint4);
```

First the starting point of a line is set (`lpPoint2`). Then with the method DPtoLP the device context units are converted into logical units. MoveTo moves to that point. After setting the end point, the LineTo command finally draws the line. This procedure is repeated 3 times, first to draw the middle line of the arrow, then the right part and finally the left part.

The following part displays diagnostic text in the top-left corner of the window: x position of subject, y position of subject, etc.

```
      pDC->SetTextColor(RGB(2, 255, 2));
      pDC->SetBkMode(TRANSPARENT);
      pDC->SetTextAlign(TA_LEFT | TA_TOP);
      itoa(subj_x,buffer,10);
      pDC->TextOut(2,0,buffer);
      itoa(subj_y,buffer,10);
      pDC->TextOut(2,20, buffer);
```

```
        itoa(dynablock_x,buffer,10);
        pDC->TextOut(42,0,buffer);
        itoa(dynablock_y,buffer,10);
        pDC->TextOut(42,20, buffer);

        itoa(delta_x,buffer,10);
        pDC->TextOut(82,0,buffer);
        itoa(delta_y,buffer,10);
        pDC->TextOut(82,20, buffer);
}
```

This ends the display procedure for the subject.

3.4 Configuration file

A configuration file is used, to be able to use the application with arbitrary scenarios. It includes the snapshot image, the width and height of the map in meters and the offset of the map compared to the scenarios starting point. Figure 2.3 shows the form and contents of this file. It is loaded from the OnInitialUpdate() method of the CBirdseyeView class by this sequence:

```
        char* pszFileName = "config.txt";
        CStdioFile myFile;
        CFileException fileException;
        if   (   !myFile.Open(   pszFileName,   CStdioFile::modeRead   ),
&fileException )
        {
              TRACE( "Can't open file %s, error = %u\n",
                    pszFileName, fileException.m_cause );
        }
        myFile.ReadString(file_name, 99);
        filename.Format("%s", file_name);
        int poz = filename.Find(10,0);      // search for " " - separator
        filename = filename.Left(poz);
        scene_x = ReadFileNum( read_buf, &myFile);  // func for reading
        scene_y = ReadFileNum( read_buf, &myFile);  // the numerical
        map_offset_x = ReadFileNum( read_buf, &myFile);   // parameters
        map_offset_y = ReadFileNum( read_buf, &myFile);
        myFile.Close();
```

The file is opened, then the first line is loaded. The application searches for the first occurrence of the space character (ASCII 10). Everything after that is considered to be the path and filename. The same procedure is applied for the numerical values too, using a separate function ReadFileNum.