User documentation: libGaze gaze tracking framework version: 1.0.0

Sebastian Herholz

June 11, 2009

Contents

1 User documentation

1.1 Intro

This document is the user documentation for the libGaze gaze tracking framework. LibGaze combines off-the-shelf eye tracking and motion capturing systems to enable gaze tracking of an unrestrained user in 3D space. The main output of this is a gaze vector in the world-coordinate system. More detailed information about how libGaze combines these systems and how its computates gaze in 3D space is described in the paper "LibGaze: Real-time gaze-tracking of freely moving observers for wall-sized displays" (http://www.kyb.mpg.de/publication.html?publ=5470).The goal of this user documentation is to explain the basic methods of using libGaze with its Python API pyGaze. The documentation contains detailed descriptions for installing the framework on Windows XP and Linux based systems. A companion tutorial follows this, which explains all of important functions pertaining to the pyGaze API, by walking the reader through a self-contained demonstration script.

1.2 Modular System

The libGaze framework uses different modules to either communicate with the different tracking systems used for eye and head tracking or to perform calculations, such as converting the raw pupil positions of the eye tracker into eye related viewing angles. A module is a dynamic C library which is loaded at runtime, by the libGaze framework. The interfaces describing the different modules are part of the libGaze package and can be found under "$/include/libGaze/interfaces/$". New modules can be written for different equipment, different calibration algorithms or different display types. A more detailed documentation about the modules and "How to create a new module" will be available in the "Developer Documentation" of libGaze, which will be included in the next release of libGaze. There are four different types of modules used by libGaze:

1. Eye tracker:
   This module communicates with the eye tracker used by libGaze and collect the raw data that it provides (e.g., screen coordinates of the pupil in the video cameras).

[0]pyGaze is a Python wrapper for the libGaze gaze tracking framework

2. Head tracker:
   This module is used to communicate with the motion tracking system to retrieve the current position of the head of the observer in the world coordinate system. Note that additional objects can also be tracked by the head tracker at the same time.

3. Calibration:
   This module contains the calibration method that is used to compute a mapping function, which specifies the relationship between the position of pupil in the eyetracker camera (in screen coordinates) and eye-in-head eccentricity (in visual angles).

4. Display:
   This module can be used to calculate intersection of the user's gaze with a 3D model of the display. It also provides functions that allows this intersection to be described in terms of the display's screen coordinates. Note that a hypothetical gaze input can also be submitted to compute the appropriate screen coordinates for stimulus rendering.

More details about the specific modules are provided Section 1.8.

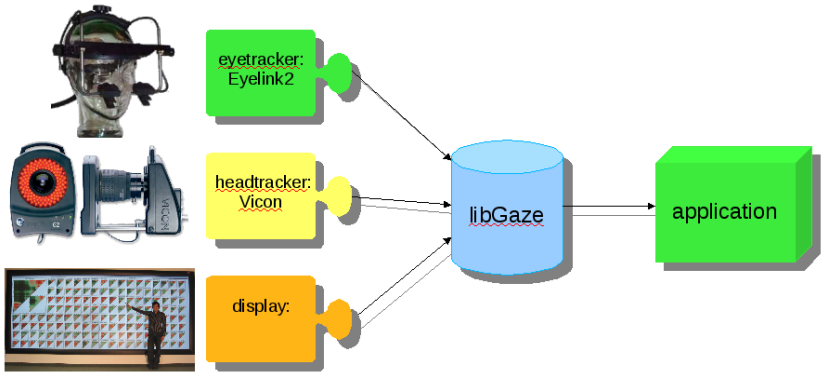

Figure 1: The libGaze framework and the relationships between different modules for: eyetracker, headtracker and display

1.3 Dependencies

The core libGaze gaze tracking framework uses a set of different open-source libraries. These libraries have to be pre-installed on the system.

Information about how to install these libraries can be found on the following websites. For Windows XP systems an additional package libGaze-1.0.0-dependencies.zip is offered containing the libraries required (see. 1.4.1).

GSL (1.8) The Gnu scientific library is used for efficient vector and matrix operations and for solving linear equations.
(http://www.gnu.org/software/gsl/)

glib (2.18) The glib framework is used to make the liGaze framework platform independent. It is used for loading dynamic libraries and for handling threads on the different supported platforms.
(http://www.gtk.org/)

parse_conf (1.0) The parse_conf library is used for parsing the INI configuration files, which specify the configuration of the different modules.
(http://sourceforge.net/projects/parseconflib/)

1.4 Installation

This section will explain how to install the python module $pyGaze$ for the libGaze tracking framework under Windows XP and under Linux.

1.4.1 Windows XP

Binary To install pyGaze you must have Python 2.5 installed on your system. You can get the latest Python 2.5 version from www.python.org. Because pyGaze depends on a set of open-source libraries (see. 1.3), additional dll's have to be installed on the system. These additional dll's are stored in the libGaze-1.0.0-dependencies.zip package which can be downloaded from the libGaze project page (http://sourceforge.net/projects/libgaze/). There are two ways of installing the additional dll's:

1. copy the dll's into the $\$SystemRoot/system32$ folder, where $\$SystemRoot$ is the folder containing Windows XP (**not recommended**)

2. storing the dll's into a folder (e.g. $C : /libGaze - 1.0.0$) and add this folder to the $\$Path$ system variable. To add a folder to the $\$Path$ variable go to $Start- > ControlPanel- > System- > Advanced- > EnvironmentVariables$. If the $Path$ variable is not listed in the $UserVariables$ list click on the New button; otherwise, select the $Path$ variable in the list and click on the $Edit$ button. Add the path to the dll folder, at the

end of the of the $Variablevalue$ field (if the $Variablevalue$ field is not empty use the ";" as a separator).

After the necessary dll's are installed. Download and execute the installation file for pyGaze (PyGaze-1.0.0-win-py2.5.zip), from the project page.

1.4.2 Linux

Binary Because libGaze uses additional libraries(see 1.3), make sure that the right version of the dependency libaries are installed on the Linux system.

Download the pyGaze package for linux (pyGaze-1.0.0-linux.tar.gz) from the libGaze project page. Extract and install the python package.

```
1 tar xf pyGaze-\lgversion-linux.tar.gz
2 cd pyGaze-\lgversion-linux
3 sudo python setup.py install
```

1.5 pyGaze Usage/Tutorial

This tutorial will explain the usage of pyGaze. It will lead the user, in stepwise manner, through all the functions that are necessary for: initializing the pyGaze framework, creating and configuring a GazeTracker object, accessing the current gaze data of an observer in realtime. More detailed information about the functions, objects and data structures of the pyGaze module can be found in the API documentation of the pyGaze module.

This tutorial also uses the VisionEgg framework for stimuli presentation and response collection (www.visionegg.org). To use VisionEgg, you have to install the following Python modules:

- Python Image Library (PIL)
- PyOpenGL
- pygame
- numpy

A guide about how to install these modules can be found on the VisionEgg webpage. A companion module (i.e., pyGazeVisionEggVisualHooks) is also provided here, that contains all the basic objects that are typically necessary for basic experimental presentation (see section 1.6). Finally, the path structure of the files used by this tutorial is shown in figure 2.

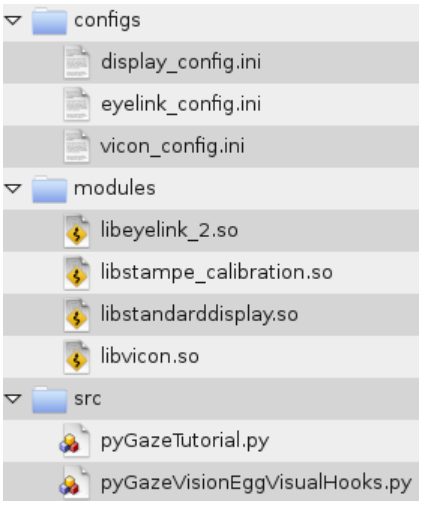

Figure 2: How the files for the tutorial are organized. The endings of the module files vary according to the used System (Windows ".dll", Linux ".so")

1.5.1 Importing modules

First, we import all the modules (including pyGaze and VisionEgg) that are going to be used in out script (line 1-5). This includes VisionEgg for visualisation.

```
1 import os,sys
2 import time
3 #modules which needs to be imported to use VisionEgg framework
4 import VisionEgg
5 VisionEgg.start_default_logging(); VisionEgg.watch_exceptions()
```

Next, we import *pyGaze* (line 8). The *pyGazeVisionEggVisualHooks* module contains an example implementation of the *IVisualisationHooks* interface called *VisionEggVisualisationHooks* (line 9). This implementation is used by pyGaze to present calibration targets and to collect user responses. A description of the implementation is given in section 1.6.

```
7  #imports the pyGaze module for
8  import pyGaze
9  from pyGazeVisionEggVisualHooks import
       VisionEggVisualisationHooks
10
11
12 #setting up graphic window
13 screen = VisionEgg.Core.get_default_screen()
14 screen.clear()
```

```
15 VisionEgg.Core.swap_buffers()
```

1.5.2 Initializing pyGaze

Next, the pyGaze framework must be initialized (line 18). During initialization, the internal timer of libGaze is reset. PyGaze can also be instructed to print debug and error messages with the two values attributed to this initialisation function (0: disabled, 1: enabled). In the example provided, we have disabled the printing of debug messages while enabling the printing of error messages.

```
17 #initializing the pyGaze framework
18 pyGaze.pyGaze(0,1)
```

1.5.3 Creation of a GazeTracker (*gt*)

Next, an instance of the pyGaze.GazeTracker class is created; namely, (*'gt'*) (line 21). Following this, we load the appropriate modules for the eye- and head-tracker (line 29). When loading these modules, the user must choose among four different modes:

pyGaze.PG_GT_BOTH Both the eye tracker and head tracker are used in combination to calculate the gaze.

pyGaze.PG_GT_EYE Only the eye tracker is used to calculate gaze and a fixed head position is assumed (position: (0.0, 0.0, 0.0), orientation: (1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0)).

pyGaze.PG_GT_HEAD Only the head tracker is used, and the eyes are always assumed to be looking straight ahead.

pyGaze.PG_GT_DUMMY None of the trackers is used. The system runs in simulation mode assuming that the eyes are looking straight ahead and that the head is not moving (position: (0.0, 0.0, 0.0), orientation: (1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0)).

These different modes allow users to test their scripts without the need to plug in a real tracking device. In our example, we load the dummy mode which allows us to run the script without any real tracking input.

```
20 #creating an GazeTracker object
21 gt = pyGaze.GazeTracker(0,1)
22
```

```
23 #locations of the used modules for eyetracker, headtracker
24 eyemod = "../modules/libeyelink_2"
25 headmod = "../modules/libvicon"
26
27
28 #loading eyetracker, headtracker and display modules
29 gt.loadModules(eyemod,headmod,pyGaze.PG_GT_MODE_DUMMY)
```

1.5.4 Configuring GazeTracker

After the modules for the eye tracker and head tracker are loaded,- they need to be configured (line 36). The configuration of the modules is specified by configuration files. Each module can have its own special configuration file format.

```
31 #locations of configurationfiles for eyetracker and headtracker
       module
32 eyecfg = "../configs/eyelink_config.ini"
33 headcfg ="../configs/vicon_config.ini"
34
35 #configuring the eyetracker and headtracker modules using
       configuration files
36 gt.configure(eyecfg, headcfg)
```

1.5.5 Loading a calibration method

A calibration module is loaded to enable calibration of the gaze tracker and to map the raw eye tracker data to viewing angles of the eye (line 41). In this example, we use a calibration module *Stampe Calibration Module* that follows a standard calibration procedure employed by most commercial eyetrackers (see 1.8.3). The calibration module

```
38 #location of the used calibration module
39 cmod = "../modules/libstampe_calibration"
40 #loading a calibration module for the eyetracker mapping
       function
41 gt.loadCalibrationModule(cmod)
```

1.5.6 Creating a VisualisationHooks object

In the next step a visualization object is created (line 53). This requires an instance of the display model, the parameters of which are submitted in

lines 48-50. The role of the visualization object is to deliver callback functions to the GazeTracker object, which allows stimuli to be presented to the user during the calibration phase. This visualization object is, therefore, submitted to the GazeTracker object (line 54). The visualization object is an instance of a class, which implements the IPyGazeVisualisationHooks interface.

```
43 #locations of the used modules for display
44 dismod = "../modules/libstandarddisplay"
45 #configuration file for the display module
46 discfg =  "../configs/display_config.ini"
47 #creating and configurating an instance of a display model
48 display = pyGaze.Display()
49 display.loadModule(dismod)
50 display.configure(discfg)
51
52 #setting the visual hooks
53 vhooks = VisionEggVisualisationHooks(screen,display)
54 gt.setVisualHooks(vhooks)
```

1.5.7 Connect GazeTracker

After the GazeTracker object is setup, a connection can be established between the eye tracker and head tracker.

```
56 #connecting to the trackers
57 gt.connect()
```

1.5.8 Open Logfile on the GazeTracker

Next, a log file is opened to record the gaze data of the observer during gaze tracking (line 60). This log file will also contain gaze data and any messages added by the user. This gaze data observes the GDF file format and can be systematically parsed by an auxiliary script (see. 1.7). To start the logging process, the method *startLogging()* is called (line 61).

```
59 #opening log files for logging data
60 gt.openLogFile("pyGazeTest","./",pyGaze.PG_GAZE_LOG_FILE)
61 gt.startLogging()
```

1.5.9 Adding messages to the log files

Any message can be printed to the log file of the GazeTracker object by using the *addLogMessage()* function.

```
88 #adds a log message to the opened log files
89 gt.addLogMessage("Start Trial")
90 #...
91 gt.addLogMessage("End Trial")
```

1.5.10 Start tracking of Gaze

After the GazeTracker object is connected to the eye- and head-tracker, the tracking process on these modules is initiated.

```
63 #starts tracking on all devices
64 gt.startTracking()
```

1.5.11 Calibrate GazeTracker

Now, it is time to calibrate the GazeTracker. The calibration procedure can be divided into three different steps.

1. **Calculate position of the eyes relative to the headtracked object:** In the first step, we want to determine the positional relationship between the head tracked object and the eyes of the observer (line 67). The system adopts a cyclopean-eye model, meaning that the positions of both the left and right eyes are assumed to originate from participant's nasal bone. During this procedure, another object, tracked by the motion-tracking system, pointed to the nasal bone and its position in the world-coordinate system is accepted by the system upon the user's keyboard input (i.e., 'spacebar'). This input comes from the IVisualisationHooks instance and has to be of the type *PG_INPUT_ACCEPT* (see section 1.6.9).

```
66 #start with the calibration procedure
67 gt.collectEyeHeadRelation()
```

2. **Align headtracked object orientation to field of view:** The second step aligns the tracked orientation of the head of the observer to the orientation of the observer's field of view (line 68). The observer is presented a rectangular shape, with a calibration target at the center. The two values given to the function correspond to the angular size of the presented rectangular object (horizontal, vertical). In our example, the rectangle has a width of 50 and a height of 40 degree. This rectangle represents the viewing frustum of the cyclopean eye and is

coupled to movements of the user's head. The orientation of the presented objects can adjusted until it is aligned with the participant's subjective field-of-view. These rotations of the viewing frustum are executed by using the following keys:

'a','y' rotates the objects around its center

's','x' shifts the objects horizontally

'd','c' shifts the objects vertically

This procedure ends when the user inputs a keyboard response of the type *PG_INPUT_ACCEPT* (e.g., 'spacebar').

```
68 gt.correctHeadDirectionVector(50,40)
```

3. **Map video image pupil positions to eye movements:** The third step computes the mapping function tht relates the eyetracker data to angular eye movements (line 70). During this, different calibration targets are presented to the observer in a random sequence. The observer should minimize head movements to ensure that the requested eye rotations are executed and fixate the presented targets with his eyes. These targets are presented at known angular distance, away from the current measured head orientation. After a fixation is detected, the calibration target disappears and another one is presented elsewhere.

   The first two values define the horizontal and vertical dimensions for the fixation grid area (in visual angles), which the mapping function should be calibrated for. The second two values define the number of calibration points (row, column) that are to be displayed. This size can depend on the used calibration module (e.g. The "Stampe Calibration Module" only works with a 3x3 grid of calibration targets 1.8.3).

   If desired, the head of the observer can be oriented to a specific position before the calibration starts (e.g. to a specific XY coordinate of a used display (line 69)). To do so, the last value describes the 3d position in the world coordinate system that the observer should orient to, before the calibration targets are presented. If *None* is passed to the function the presentation of calibration targets starts immediately, without doing a head orientation fixation first. In our example, 'dpos' is calculated from the coordinates of our display's screen center (line 69).

```
69 dpos = display.get3DPositionFromXY(512,384)
70 gt.calibrate(50,40,3,3,dpos)
```

1.5.12 Validate GazeTracker calibration

After the GazeTracker object is calibrated, the mapping function can be validated using the *validateCalibration()* function. This process is the same as step 3 of the calibration process. This time the presented fixation targets are used to calculate the differences between the presented target locations and the calculated gaze using the fitted mapping function. The result of the validation is saved in an instance of the ValidationDataSet class. This class contains information about the accuracy of the GazeTracker calibration.

```
72 #validate the calibration
73 vDataSet = pyGaze.ValidationDataSet()
74 gt.validateCalibration(50,40,3,3,dpos,vDataSet)
75 print "Average validation error:", vDataSet.avg_drift
```

The result of the last executed validation can also be visualized on display using the $displayLastValidation()$ function. This function shows the positions of the gaze fixations, as computed by the mapping function, and the position of the presented validation targets.

```
76 gt.displayLastValidation()
```

1.5.13 Get Gaze

The $getCurrentGaze()$ function allows the user to access the current gaze of the observer (line 84). This function takes as parameter an instance of $GazeDataSet$ (line 83), in which the gaze data about the left and the right eye of the observer will be stored. The gaze data corresponds to the world coordinate system.

```
82 #getting the current gaze of the user
83 gds = pyGaze.GazeDataSet()
84 gt.getCurrentGaze(gds)
85 print "Gaze Data for the left eye:", gds.gaze[0].origin,gds.gaze
       [0].direction
86 print "Gaze Data for the right eye:", gds.gaze[0].origin,gds.
       gaze[0].direction
```

1.5.14 Drift Corrections

During tracking, periodic checks can be executed to determine if the calibration of the system is still valid. ???

1.5.15 Stop tracking of Gaze

Once all data is recorded, the tracking of gaze and the tracking of the eye tracker and head tracker can be stopped by calling $stopTracking()$ from the GazeTracker object (i.e. gt).

```
93 #stopping the tracking of gaze data
94 gt.stopTracking()
```

1.5.16 Close Log-file

Next, the corresponding log file should be closed by calling the $closeLogFile()$ method of the GazeTracker object (i.e. gt).

```
96 #stops logging tracked data into the log files
97 gt.stopLogging()
```

1.5.17 Disconnect Gazetracker

Finally all connections to the eyetracker and headtracker should be disconnected.

```
102 #disconnects all tracker
103 gt.disconnect()
```

1.6 IVisualisationHooks

```
1  import os, sys
2  import VisionEgg
3  VisionEgg.start_default_logging()
4  VisionEgg.watch_exceptions()
5  from VisionEgg.Core import *
6  from VisionEgg.MoreStimuli import Target2D
7  from VisionEgg.Text import Text
8  from VisionEgg.Textures import Texture, TextureStimulus
9  from VisionEgg.MoreStimuli import *
10
11 import pygame
```

```
12 from pygame.locals import *
13 from numpy import *
14
15 import pyGaze
16 from pyGaze import PG_VALID_TARGET,PG_VALID_LEFT,PG_VALID_RIGHT
17 import time
18
19 class VisionEggVisualisationHooks:
20         screen = None
21         resolution = None
22
23         viewport = None
24
25         display = None
26
27         def __init__(self,screen, display):
28             self.screen = screen
29             self.display = display
30             self.resolution =screen.size
31
32         def __del__(self):
33             print "del"
```

1.6.1 eraseScreen

Clears all content presented on the screen.

```
1         def eraseScreen(self):
2             self.screen.parameters.bgcolor = (0.5,0.5,0.5,1.0)
3             self.screen.clear()
```

1.6.2 updateScreen

Updates screen content. Elements which were drawn after the last $eraseScreen()$ are displayed.

```
1         def updateScreen(self):
2             swap_buffers()
```

1.6.3 drawCalTarget

Draws a calibration target on the intersection point of a 3D viewing ray with the display. The ray is represented by its origin and direction in the

world coordinate system. First, the 3D position of this intersection is calculated and then, its screen coordinates are calculated. Finally, the calibration target is drawn at the calculated coordinates.

```
1   def drawCalTarget(self,origin,direction):
2               #calculates 3D intersection point with the display
3               xy3d = self.display.getIntersectionWithScreen(origin
                    ,direction)
4               #gets display coordinates from intersection
5               xy = self.display.getXYPositionFrom3D(xy3d)
6               x = xy[0]
7               y = xy[1]
8               #draws calibration target at display coordinates
9               wcircle = FilledCircle(anchor   = 'center',
10                position = (x,self.resolution[1]-y),
11                radius   = 6.0, color     = (255, 255, 255),
12                anti_aliasing = True)
13              bcircle = FilledCircle(anchor   = 'center',
14                position = (x,self.resolution[1]-y),
15                radius   = 3.0, color     = (0, 0, 0),
16                anti_aliasing = True)
17              viewport = Viewport( screen=self.screen,
18                stimuli=[wcircle,bcircle] )
19              viewport.draw()
20              return xy3d
```

1.6.4 drawValidation

Draws the validation data. The ray is represented by its origin and direction in the world coordinate system. The *type* parameter is used to specify if the ray corresponds to the presented validation target or to the caluated gaze of the left or right eye. Depending on the type a different color is choosed.

```
1           def drawValidation(self,origin, direction,type):
2               r = 8
3               #ray corresponds to the presented validation target
4               if type == PG_VALID_TARGET:
5                   colorRGBA = (255,0,0,255)
6               #ray corresponds is the calculated gaze for the left
                    eye
7               elif type == PG_VALID_LEFT:
8                   colorRGBA = (0,255,0,255)
9               #ray corresponds is the calculated gaze for the
                   right eye
10              elif type == PG_VALID_RIGHT:
11                  colorRGBA = (0,0,255,255)
```

```
12              #calculates 3D intersection point with the display
13              xy3d = self.display.getIntersectionWithScreen(origin
                    ,direction)
14              #gets display coordinates from intersection
15              xy = self.display.getXYPositionFrom3D(xy3d)
16              x = xy[0]
17              y = xy[1]
18
19              #draws the validation object
20              circle = FilledCircle(anchor   = 'center',
21            position = (x,self.resolution[1]-y),
22            radius   = r, color     = colorRGBA)
23              viewport = Viewport( screen=self.screen, stimuli=[
                    circle] )
24              viewport.draw()
25              return xy3d
```

1.6.5 drawFov

Draws the approximate position and orientation of the field of view of the observer. This position is represented by four rays that represent each corner of the approximated position of the FOV. A ray is represented by a list of two elements. The first elements is the origin of the ray and the second element is the direction vector of the ray. Both elements are 3D vectors in the world coordinate system.

```
1           def drawFov(self,r0,r1,r2,r3):
2
3               RGBA=(255,0,0,255)
4               xy3d0 = self.display.getIntersectionWithScreen(r0[0]
                    ,r0[1])
5               xy0 = self.display.getXYPositionFrom3D(xy3d0)
6
7               xy3d1 = self.display.getIntersectionWithScreen(r1[0]
                    ,r1[1])
8               xy1 = self.display.getXYPositionFrom3D(xy3d1)
9
10              xy3d2 = self.display.getIntersectionWithScreen(r2[0]
                    ,r2[1])
11              xy2 = self.display.getXYPositionFrom3D(xy3d2)
12
13              xy3d3 = self.display.getIntersectionWithScreen(r3[0]
                    ,r3[1])
14              xy3 = self.display.getXYPositionFrom3D(xy3d3)
15              z=0
16              rect = Rectangle3D(vertex1= (xy3[0],self.resolution[
                    1]-xy3[1],z),
```

```
17              vertex2= (xy1[0],self.resolution[1]-xy1[1],z),
18              vertex3= (xy0[0],self.resolution[1]-xy0[1],z),
19              vertex4= (xy2[0],self.resolution[1]-xy2[1],z),
20              color = (0.0,1.0,0.0,1.0))
21                viewport = Viewport( screen=self.screen, stimuli=[
                    rect] )
22                viewport.draw()
```

1.6.6 drawHeadOrientationTarget

Draws the head orientation target where a ray in the world coordinate system intersects the display model.

```
1           def drawHeadOrientationTarget(self,x, y,r,RGBA =(0.0,0.0
                ,0.5,1.0) ):
2                circle = FilledCircle(anchor    = 'center',
3             position = (x,self.resolution[1]-y),
4             radius    = r, color     = RGBA)
5                viewport = Viewport( screen=self.screen, stimuli=[
                    circle] )
6                viewport.draw()
```

1.6.7 drawHeadOrientation

Draws the current head orientation of the observer. The current orientation is represented by a ray in the world coordinate system.

```
1
2           def drawHeadOrientation(self,x, y,r):
3                circle = FilledCircle(anchor    = 'center',
4             position = (x,self.resolution[1]-y),
5             radius    = r,
6             color     = (0.5,0.0,0.0,1.0))
7                viewport = Viewport( screen=self.screen, stimuli=[
                    circle] )
8                viewport.draw()
```

1.6.8 showMessage

Prints a message that is displayed on the screen (e.g. calibration instructions).

```
1           def showMessage(self,msg):
2                size=20
3                text = Text(text=msg, anchor='center',
```

```
4            position=(self.resolution[0]/2.0,self.resolution[1]/2.
                 0),
5            color=(255,255,255))
6              viewport = Viewport( screen=self.screen, stimuli=[
                   text] )
7              viewport.draw()
```

1.6.9 getInput

Returns the current input by the user (e.g. keypress).

```
1          def getInput(self):
2              input = pyGaze.Input()
3              #sets the input type to INPUT_NONE when no input was
4          #made by the user
5              input.type = pyGaze.PG_INPUT_NONE
6              eventq = pygame.event.get([pygame.KEYDOWN])
7
8              for event in eventq:
9
10                 if event.type == pygame.KEYDOWN:
11                     pygame.event.clear()
12                     #if the space key was pressed the input type
                           is ACCEPT
13                     if event.key == pygame.K_SPACE:
14                         input.type =  pyGaze.PG_INPUT_ACCEPT
15                         break
16                    #if the return key was pressed the input type
                         is ABORT
17                     elif event.key == pygame.K_RETURN:
18                         input.type =  pyGaze.PG_INPUT_ABORT
19                         break
20                     #if the ESC key was pressed the input type is
                           QUIT
21                     elif event.key ==pygame.K_ESCAPE:
22                         input.type =  pyGaze.PG_INPUT_QUIT
23                         break
24                     #if any other key was pressed the input type
                           is KEY
25         #and the pressed key is saved under "value"
26                     else:
27                         input.type = pyGaze.PG_INPUT_KEY
28                         input.key = event.key
29                         break
30
31             pygame.event.clear()
32             return input
```

1.6.10 clearInput

Clears the current Input queue.

```
1         def clearInput(self):
2             pygame.event.clear()
```

1.7 GDF file format

The *.gdf file format contains the raw tracked gaze data that are recorded by the libGaze gaze tracking framework as well as some additional information (user messages, validation information etc.). This is a text file that contains tab separated values. A row begins with a character specifying the type of data presented in this row, followed by a timestamp for when the data set was generated. Each row in the .gdf file can be one of four following types.

Gaze data A gaze data set contains the tracked gaze of the observer. It begins with the character "G" followed by processed gaze, eye and head data.

gaze data The gaze data contains 14 different values (gTime, gType, gLPX, gLPY, gLPZ, gLDX, gLDY, gLDZ, gRPX, gRPY, gRPZ, gRDX, gRDY, gRDZ) from column 2 to column 15.

gTime: The time in milliseconds when the gaze data set was calculated by the libGaze tracking framework. The time is relative to the initialization of the framework.

gType: The eye the following gaze data is valid (1 = both eyes, 2 = left eye, 3 = right eye, 4 = none of the eyes). This data flag can be used to identify blinks.

gLPX, gLPY, gLPZ: The 3D XYZ postion of the gaze origin for the left eye in the world coordinate system.

gLDX, gLDY, gLDZ: A normalized 3D direction vector for the gaze direction of the left eye, relative to the gaze origin in the world coordinate system.

gRPX, gRPY, gRPZ: The 3D XYZ postion of the gaze origin for the right eye in the world coordinate system.

gRDX, gRDY, gRDZ: A normalized 3D direction vector for the gaze direction of the right eye, relative to the gaze origin in the world coordinate system.

eye data The eye data contains 8 values (eTime, eType, eLAlpha, eLBeta, eLSize, eRAlpha, eRBeta, eRSize) from column 16 to column 24.

- **eTime** The time (milliseconds) when the original raw eye data was received from the eye tracker.
- **eType** The eye for which data is valid (1 = both eyes, 2 = left eye, 3 = right eye, 4 = none of the eyes).
- **eLAlpha** The horizontal viewing angle in degrees in the eye related coordinate system for the left eye. In other words, the azimuth rotation of the left eye.
- **eLBeta** The vertical viewing angle in degrees in the eye related coordinate system for the left eye. In other words the elevation rotation of the left eye.
- **eLSize** The size of the left pupil in eye trackers camera image in image pixels.
- **eRAlpha** Same as eLAlpha but for the right eye.
- **eRBeta** Same as eLBeta but for the right eye.
- **eRSize** Same as eLSize but for the right eye.

head data The head data contains 13 values (hTime, hPX, hPY, hPZ, hO00, hO01, hO02, hO10, hO11, hO12, hO20, hO21, hO22) from column 25 to column 38.

- **hTime:** The time (milliseconds) when the raw head data was received from the head tracker.
- **hPx, hPY, hPZ:** The 3D XYZ position of the head (nasal bone) of the observer in the world coordinate system.
- **hO00, ..., hO22:** The 3x3 orientation matrix of the observer's head, in the world coordinate system.

Messages A message data data row contains messages added by the user. It starts with the character "M" followed by a timestamp telling when the message was added to the ".gdf" file. After the timestamp the user message is added as a string.

Validation The validation data set contains information about a validation of the gaze tracker's calibration. It starts with the character "V"

followed by a timestamp when the validation was proceeded. It contains 8 values (vAVGL, vAVGR, vMAXL, vMAXR, vAVGHL, vAVGHR, vAVGVL, vAVGVR).

vAVGL, vAVGR: The average drift in degrees for the left and right eye.

vMAXL, vMAXR: The maximum drift in degrees for the left and right eye.

vAVGHL, vAVGHR: The average horizontal drift in degrees for the left and right eye.

vAVGVL, vAVGVR: The average vertical drift in degrees for the left and right eye.

DriftCheck The drift check data set contains information about a check performed to check for drifts of the gaze tracker equipment. It starts with the character "D" and the values are the same as the validation data.

1.7.1 PyGazeGDFParser

To extract additional data (like xy display coordinates) out of a GDF file, it is possible to use the python class PyGazeGDFParser. To calculate screen coodinates from the given gaze data of the GDF file, the $pyGazeGDFParser$ needs a model of the display. This display model describes the display's geometry in the 3D world coordinate system and allows intersections of the observers gaze and the display model to to be calculated. During the parsing process, all data which is not marked as gaze data will be copied one to one to the new output file (e.g. user messages or validation informations). All lines marked as gaze data will be used to calculate the additional data. To specify which additional data should be calculated from the given gaze data the user has to set specific parse options.

```
1  import pyGaze
2  from pyGazeGdfParser import *
3
4  #initializing the pyGaze framework
5  pyGaze.pyGaze()
6
7  #loading and configuring a display model
8  dismod = "modules/libstandarddisplay"
9  discfg = "configs/display_config.ini"
10 display = pyGaze.Display()
11 display.loadModule(dismod)
```

```
12 display.configure(discfg)
13
14 #creating an instance of the PyGazeGdfParser
15 gp = PyGazeGdfParser()
16 gp.loadGDF("../pyGazeTest.gdf")
17
18 #setting the display attribute of the parser
19 #to be the loaded display module
20 gp.setDisplay(display)
21 gp.setParseOption(PyGazeGdfParser.DISPLAY_GAZE | PyGazeGdfParser
       .DISPLAY_HEAD)
22 #parsing the loaded GDF file and saving
23 #the calculated results into an new file
24 gp.parseGDF("../pyGazeTest.gaze")
```

Parse options The PyGazeGDFParser can use different parse options to specify the data that should be calculated from a given gaze data set of the GDF file. The different options can be combined using an "—" operator. Each parse option corresponds to a specific set of output data. The order of the output data is fixed (DISPLAY_GAZE, GAZE, EYE, DISPLAY_HEAD, HEAD). Only datasets that are specified will be parsed. In our example we specify for only DISPLAY_GAZE and DISPLAY_HEAD to be prcessed.

The first 3 values are character "G" to identify that the following data is a gaze data set, a timestamp and the identifier for which eye the following gaze and eye data is valid.

DISPLAY_GAZE: The display gaze data set contains 4 values (dXL, dYL, dXR, dYL) which represent the xy display coordinates for the left and right eye.

GAZE: The gaze data set contains 12 values (gLPX, gLPY, gLPZ, gLDX, gLDY, gLDZ, gRPX, gRPY, gRPZ, gRDX, gRDY, gRDZ) representing the gaze origins and directions for the left and right eye.

gLPX, gLPY, gLPZ: The 3D XYZ postion of the gaze origin for the left eye in the world coordinate system.

gLDX, gLDY, gLDZ: A normalized 3D direction vector for the gaze direction of the left eye relative to the gaze origin in the world coordinate system.

gRPX, gRPY, gRPZ: The 3D XYZ postion of the gaze origin for the right eye in the world coordinate system.

gRDX, gRDY, gRDZ: A normalized 3D direction vector for the gaze direction of the right eye relative to the gaze origin in the world coordinate system.

EYE: The eye data contains 6 values (eLAlpha, eLBeta, eLSize, eRAlpha, eRBeta, eRSize) representing the horizontal and vertical viewing angles and the size of the left and right eye.

eLAlpha The horizontal viewing angle in degrees in the eye related coordinate system for the left eye.

eLBeta The vertical viewing angle in degrees in the eye related coordinate system for the left eye.

eLSize The size of the left pupil in eye trackers camera image. The unit should be image pixels.

eRAlpha The horizontal viewing angle in degrees in the eye related coordinate system for the right eye.

eRBeta The vertical viewing angle in degrees in the eye related coordinate system for the right eye.

eRSize The size of the right pupil in eye trackers camera image. The unit should be image pixels.

DISPLAY_HEAD: The display head data set contains 2 values (dXH dYH) representing display coordinates calculated using the position and orientation of the head to calculate an intersection with the display.

HEAD: The head data set contains 6 values (hPX, hPY, hPZ, hEulerX, hEulerY, hEulerY) representing the position of the head (nasal bone) and its orientation given by 3 Euler angles.

ALL: If the parse option "ALL" is set all other option are enabled.

1.8 Modules

This section describes the four different module types used by the libGaze framework.

1.8.1 Eye tracker modules

Eye tracker modules are used to communicate with the specific eye tracker. Because there are different eyetrackers on the market, each with different APIs, an eye tracker module has to be programmed for each different

eye tracker. This module wraps the eye tracker specific API call to our generic set of libGaze module functions (e.g connect(), disconnect(), ...). A C-header file containing all needed function declarations for an eye tracker module can be found under:
"$interfaces/libGaze\_eye\_tracker\_module.h$" Also, the format of the returned eye tracking data is standardized to the XY pupil positions of the eyes in the eye tracker's camera images and the size of the pupils in the image.

Eyelink 2 The eyelink 2 eye tracker module is a wrapper for the EyeLink2 eye tracker from SR-Research (www.sr-research.com).

installation To use the eyelink 2 eye tracker module the EyeLink2 API package containing the driver for the Eyelink2 from SR-Research has to be installed on the system. The API can be found in the support forum of SR-Research (https://www.sr-support.com/forums/) under $Downloads->EyelinkDisplaySoftware$.

Windows: From the forum topic $WindowsDisplaySoftware$, download the EyeLink Developer For Windows Version 1.8.1 (EyeLinkDevKit_win32_1.8.1.zip). Extract the file and run the installer. Install the package to a specific folder like $C:/SRResearch/Eyelink$. To make the Eyelink2 API accesible for the module add the $libs$ folder of the package (e.g. $C:/SRResearch/Eyelink/libs$) to the $\$Path$ system variable (see 1.4.1).

Linux: From the topic $LinuxDisplaySoftwarePackage$ download the EyeLink Linux Display Software (EyeLinkDisplaySoftwareLinux1.7.276.tar.gz). Extract the packge and copy the folders $/bin$, $/lib$ and $/include$ to $/usr/local/$.

```
1 tar xf EyeLinkDisplaySoftwareLinux1.7.276.tar.gz
2 cd EyeLinkDisplaySoftwareLinux1.7.276
3 sudo cp -r bin /usr/local
4 sudo cp -r lib /usr/local
5 sudo cp -r include /usr/local
```

An example of a configuration file, which configures the eyelink2 eye tracker module can be found in the eyelink2 module package ("$cfg/eyelink\_config.ini$").

1.8.2 Head tracker modules

Head tracker modules are used to communicate with the specific motion capturing system, which reveals the user's position and orientation in a world coordinate system. As with eyetrackers, a different module has to be written to use a specific motion captureing system. The module acts as a wrapper between the specific API call and a generic set of module functions (e.g connect(), disconnect(), ...). A C-header file containing all needed function declarations for a head tracker module can be found under: "$interfaces/libGaze\_head\_tracker\_module.h$" Also the format of the returned head tracking data is standardized to a 3D postion of the observer in the world coordinate system and a right handed rotation matrix representing the current orientation of the observer in the world coordinate system.

Vicon The vicon head tracker module is a wrapper for motion capturing systems from Vicon (www.vicon.com) for their Vicon IQ 2.5 tracking software. Because the Vicon IQ 2.5 tracking software communicates over a TCP/IP protocol, no additional software has to be installed on the system. An example of a configuration file, which configures the vicon head tracker module can be found in the vicon module package ("$cfg/vicon\_config.ini$").

1.8.3 Calibration modules

The calibration modules provide functions to calibrate the mapping of the XY pupil positions in the camera images of the used eye tracker to the horizontal and vertical viewing angles in an eye-in-head coordinate system. Because there are several ways to calculate a mapping between the pupil positions and the eye related viewing angles it is possible to implement and use different methods. A C-header file containing all needed function declarations for a calibration module can be found under: "$interfaces/libGaze\_calibration\_module.h$"

Stampe The stampe calibration module uses an algorithm presented by Stampe (1993) . The calibration grid used to calibrate the algorithm has to be of the size of 3x3.

[0] "Heuristic filtering and reliable calibration methods for video-based pupil-tracking systems"

1.8.4 Display modules

The display modules of libGaze provide functions to calculate intersections with the input gaze of an observer and a 3D representation of a display in the world coordinate system. These intersections are represented by the display coordinates which contain the 2D XY display coordinate of the display and the 3D position of this XY display coordinate in the world coordinate system. Because it is possible to use different kinds and shapes of displays, different displays modules can be implemented (e.g. planar, cureved or grid displays). A C-header file containing all needed function declarations for a display module can be found under "$interfaces/libGaze\_display\_module.h$" An example of a configuration file, which configures the planar display module can be found in the planar display module package ("$cfg/planar\_display\_config.ini$").

Planar The display module represents a standard planar display in the world coordinate system.

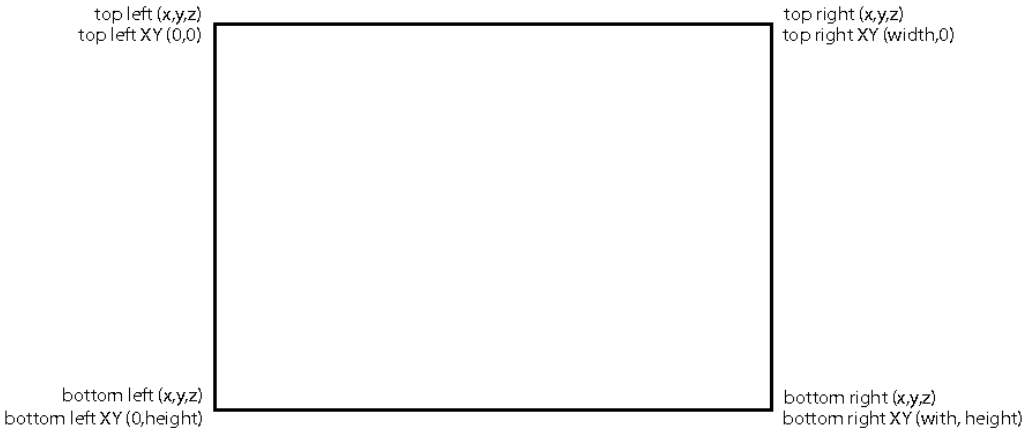

Figure 3: Example of a planar display configuration where the origin of the display coordinate system is in the top left corner.