et_simul – A Framework for Simulating Eye Trackers

Martin Böhme
University of Lübeck, Germany
boehme@inb.uni-luebeck.de

1 Introduction

This is et_simul, a framework for simulating eye trackers – in particular, the gaze estimation step. For a paper describing the framework and some experiments performed using it, see [1].

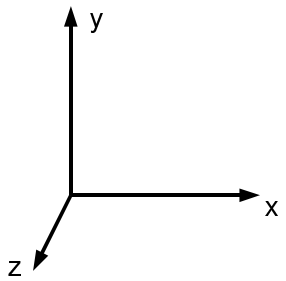
To get started, start `interpolate_test.m` (which tests a simple interpolation-based algorithm) from a Matlab command line.

The project uses a pseudo-object-oriented philosophy, i.e. Matlab structures are used to represent the various objects in the system (eyes, lights, cameras), and methods are implemented as functions that take the object they operate on as their first argument. The name of a method should start with the name of the type of object it operates on (e.g. `eye_look_at()`, `camera_project()`); methods that create objects ("constructors") have the suffix `_make` (e.g. `eye_make()`, `camera_make()`).

2 Geometric conventions

The following conventions are used in geometrical calculations:

- All points and vectors are represented in homogeneous coordinates
- All vectors are column vectors
- All coordinate systems are right-handed:



Note: For a camera whose image plane is the x-y-plane, this would mean that its optical axis points along the *negative* z-axis.

- All measurements are in metres
- Object transformation matrices always transform object coordinates to world coordinates.

  Rationale: Consider a transformation matrix of the following form

$$\begin{pmatrix} \ddots & & \iddots & \vdots \\ & A & & d \\ \iddots & & \ddots & \vdots \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

  Then d is just the position of the object in world coordinates; and to rotate an object around the centre of its local coordinate system by a matrix B, we just concatenate B onto A.

3 A Short Example

The following short example is designed to introduce some of the functions in the framework. The source files for the functions contain more detailed documentation.

The example code can also be run using the file `example.m`.

To begin with, we will run a test of a simple interpolation-based gaze estimation method:

```
interpolate_test();
```

The test requires a few moments to run; when it is finished, it shows a plot that visualizes the relative magnitude and direction of the error at different positions of the screen. Note that the size of the error arrows is not to the same scale as the screen coordinates.

Eye, light, and camera objects

As mentioned above, the framework follows an object-oriented philosophy. An eye object, for example, is created like this:

```
e=eye_make(7.98e-3, [1 0 0; 0 0 1; 0 1 0]);
```

The eye has a corneal radius of 7.98 mm and its optical axis points along the negative y-axis. (In the eye's local coordinate system, the optical axis points along the negative z-axis. By specifying an eye-to-world transformation matrix that exchanges the y- and z-axis, we make the optical axis of the eye point along the negative y-axis of the world coordinate system.)

We now position the centre of the eye at $(x, y, z) = (0, 0.5, 0.2)$ (all coordinates in metres):

```
e.trans(1:3, 4)=[0 500e-3 200e-3]';
```

Note that we use the subscript `(1:3, 4)` to access the position vector in the transformation matrix (denoted by d in the previous section).

Next, we will create a light and set its position to (0.2, 0, 0):

```
l=light_make();
l.pos=[200e-3 0 0 1]';
```

Because lights are modelled as perfect point light sources, they do not have an orientation, and hence they do not need a full transformation matrix; only the position has to be specified.

We also create a camera:

```
c=camera_make();
```

In its local coordinate system, the camera points out along the negative z-axis. We want to change the camera's orientation so that it points along the positive y-axis, towards the eye:

```
c.trans(1:3,1:3)=[1 0 0; 0 0 -1; 0 1 0];
```

By default, the camera is positioned at the origin of the world coordinate system; we leave this default unchanged.

Visualizing an eye tracking setup

We can now visualize our eye tracking setup:

```
draw_scene(c, l, e);
```

This draws a three-dimensional representation of the following:

- The camera (the camera's view vector and the axes of its image plane)
- The light (shown as a red circle)
- The eye (showing the surface of the cornea, the pupil centre, the cornea's centre of curvature, and the CRs)

Cell arrays containing more than one eye, light, or camera may also be passed to `draw_scene`.

Calculating positions of CRs

We now wish to calculate the position of the corneal reflex in space, defined as the position where the ray that emanates from the light and is reflected into the camera strikes the surface of the cornea:

```
cr=eye_find_cr(e, l, c);
```

We can now determine the position of the CR in the camera image:

```
cr_img=camera_project(c, cr);
```

In reality, the position of features in a camera image cannot be determined with infinite accuracy. To model this, a so-called *camera error* can be introduced. This simply causes `camera_project` to offset the point in the camera image by a small random amount. For example, the following specifies a Gaussian camera error with a standard deviation of 0.5 pixels:

```
c.err=0.5;
c.err_type='gaussian';
```

Note that the camera error is a property of the camera. By default, the camera error is set to zero.

Eye tracker object

An eye tracker is represented by an eye tracker object, which has the following properties:

- A cell array `cameras` of one or several camera objects.
- A cell array `lights` of one or several light objects.
- A matrix `calib_points` of calibration points.
- A function handle `calib_func` for the calibration function, which is supplied with the observed positions of the pupil centre and CRs for every calibration point and uses this information to calibrate the eye tracker.

- A function handle `eval_func` for the evaluation function, which is used to perform gaze measurements after calibration. It is supplied with the observed positions of the pupil centre and CRs and outputs the gaze position on the screen.

For more information, see the documentation in `et_calib`.

The function `interpolate_make` creates an eye tracker that uses a simple interpolation-based gazed estimation scheme. The eye tracker can be tested directly using the test harness `test_over_screen`:

```
test_over_screen(interpolate_make());
```

4 Functions

This function summarizes the most important functions in the framework. Detailed documentation for the functions can be found in the Matlab source files.

Eye functions

`eye_make`	Creates a structure that represents an eye
`eye_draw`	Draws a graphical representation of an eye
`eye_find_cr`	Finds the position of a corneal reflex
`eye_find_refraction`	Computes observed position of intraocular objects
`eye_get_pupil_image`	Computes image of pupil boundary
`eye_get_pupil`	Returns an array of points describing the pupil boundary
`eye_look_at`	Rotates an eye to look at a given position in space
`eye_refract_ray`	Computes refraction of ray at cornea surface

Light functions

`light_make`	Creates a structure that represents a light
`light_draw`	Draws a graphical representation of a light

Camera functions

`camera_make`	Creates a structure that represents a camera
`camera_draw`	Draws a graphical representation of a camera
`camera_pan_tilt`	Pans and tilts a camera towards a certain location
`camera_project`	Projects points in space onto the camera's image plane
`camera_take_image`	Computes the image of an eye seen by a camera
`camera_unproject`	Unprojects a point on the image plane back into 3D space

Optical helper functions

`reflect_ray_sphere`	Reflects ray on sphere
`refract_ray_sphere`	Refracts ray at surface of sphere

`find_reflection`	Finds position of a glint on the surface of a sphere
`find_refraction`	Computes image produced by refracting sphere

Test harnesses

`test_over_observer`	Computes gaze error at different observer positions
`test_over_screen`	Computes gaze error at different gaze positions on screen

Preimplemented gaze estimation algorithms

`beymer_*`	Method of Beymer and Flicker (2003)
`interpolate_*`	Simple interpolation-based method
`shihwuliu_*`	Method of Shih, Wu, and Liu (2000)
`yoo_*`	Method of Yoo and Chung (2005)
`hennessey_*`	Method of Hennessey, Noureddin, and Lawrence (2006)

5 Legal Notice

References

[1] Martin Böhme, Michael Dorr, Mathis Graw, Thomas Martinetz, and Erhardt Barth. A Software Framework for Simulating Eye Trackers. *Proceedings of Eye Tracking Research & Applications (ETRA)*, 2008 (to appear).