ECL, version 0.9 – September 2009

Introduction

The Dual Purkinje Image (DPI) eye tracker has enjoyed decades of popularity as a tool in eye movement research. Though many high-quality digital eye tracking systems have been developed to offer wider applicability, better ease of use and smoother integration with existing computing platforms, the DPI, which does not come with any software, continues to occupy a valuable niche in the market due to its analog nature and great spatial and temporal precision.

In this report, we describe the functionality of the Eyetracker Code Library (ECL), a software library that improves upon existing DPI software for the DOS platform (Van Rensbergen & De Troy, 1993; Van Diepen, 1998), enabling researchers to program experiments on a Windows XP or Vista platform using the C++ programming language. Features include the measurement and calibration of the eye movement signal, online saccade or fixation detection, drift correction, support for a control monitor, output to a custom file format, and a modular object-oriented design that allows flexible adaptation and extensibility of ECL to the researcher's specific hardware configuration and scientific needs.

In particular, ECL is built around five main classes:

- **ecl_daq,** which implements functionality to acquire raw eye movement data. Typically this is done through analog-to-digital conversion of the DPI output signal, but alternate data sources are also possible. For instance, the computer mouse can be used for simulation purposes.
- **ecl_process**, which implements functionality to perform a calibration of the raw data and convert them to real-time pixel coordinates. In addition, the current status of the eye is returned (for instance, 'in saccade').
- **ecl_output**, which implements functionality to save the full eye movement and eye status data, as well as experimental events and comments, to a file on the computer.

- **ecl_monitor**, which implements functionality to perform basic operations onto a monitor, be it the experimental monitor or a control monitor.
- **ecl_response**, which implements functionality to collect a manual response from devices such as a keyboard or a response box.

In the tradition of object-oriented design, most of these base classes declare abstract functions that are to be implemented through the definition of child classes. For instance, ecl_response declares a function is_pressed(), without defining an implementation. Its child class ecl_response_keyboard contains a keyboard-specific implementation of is_pressed(), while child class ecl_response_mouse contains an implementation for a computer mouse button. The crux of the matter now is that these objects can be passed to other functions as if they were objects of the base type, of which the abstract function calls will be automatically mapped onto the appropriate child class function call. To continue the example, ecl_process.calibrate(ecl_response * rp) requires a pointer to an ecl_response object to collect binary responses for its calibration routine; both ecl_response_mouse and ecl_response_keyboard will be accepted as valid ecl_responses, as will any other custom child class object that implements the appropriate functions. This approach has two main advantages. First, ECL experiments can be flexibly ported between various hardware setups. Changing a single line of code can be sufficient to change the response device, monitor, output file format or eye movement data acquisition device throughout the entire experiment. Second, currently unsupported hardware devices can be added by the user without needing to meddle in the inner workings of ECL. When discussing ECL functionality, we will describe base classes in detail and their child classes, which may contain additional functions, in more general terms.

The DPI's output

The DPI eye tracker delivers four analog output signals, with a potential range of -5.12V to +5.12V each. ***H*** and ***V*** signal the gaze position in the horizontal and the vertical direction, respectively. Ideally, a fixation on the center of the experimental monitor corresponds to a double 0V signal, whereas gaze positions that deviate from the center of the screen allow the signals to vary across their full range. ***B*** on the other hand is a de facto binary signal indicating that the subject is blinking his eye, set to either 0V (no blink) or +5.12V (blink). The ***T*** signal, similarly, indicates whether the DPI machine is successfully tracking the eye position. 0V signals that the track has been lost, +5.12V that the measurement is valid.

Since these signals are analog they do not suffer from processing delays or the temporal resolution restrictions that many digital machines do exhibit. However, some of this will be lost through the necessary digital-to-analog conversion that ecl_daq will perform. Experimenters should therefore always be aware of the limitations inherent to their specific ecl_daq implementation, as this will be the least visible bottleneck in achieving real-time eye position measurements. Delays induced by further steps in the processing of the signal and the presentation of visual stimulation, on the other hand, will lead to visible time-out errors in the output of ECL.

For further information on using the DPI eye tracker, we refer to the DPI operation manuals.

ECL library classes

ecl_daq

Description

Each experiment uses an ecl_daq (short for Data AcQuisition) object as a bridge between the analog eye tracker signals and ecl_process expecting these signals to be expressed as numerical variables in C++. We provide implementations for three different hardware solutions, as well as a mouse-driven and a dumpfile-driven method. Ofcourse, the user is free to create additional ecl_daq child classes compatible with his or her own specific hardware or research needs.

To obtain raw samples, the user will typically not need to use the ecl_daq object directly. The ecl_process object used will simply take an ecl_daq as an argument and convert its raw data values to pixel coordinates on the screen, as well as determine the current eye movement status. For debugging or testing purposes, however, direct access can be useful.

Note that ECL is designed to operate with a temporal resolution of 1000Hz. As a result, the ecl_daq functionality we provide only updates the current eye position sample once every millisecond. More frequently repeated attempts to retrieve the eye position, though not forbidden, will not result in updated raw data samples until the next millisecond period commences.

Base class definition

```
                ecl_DAQ();
virtual        ~ecl_DAQ();

virtual bool init()=0;
virtual bool is_initialised()=0;
virtual bool start_sampling()=0;
virtual bool stop_sampling()=0;
virtual bool is_sampling()=0;
virtual bool get_samples(long tstamp, long &h, long &v, long &b, long &t)=0;
virtual bool monitor(ecl_monitor * mon, ecl_response * rp)=0;
```

init() contains the initialisation routines specific to the data acquisition method. This function should be called after the creation of an ecl_daq object, but before passing it on to ecl_process. It is, like all ECL functions, a *synchronous* or *blocking* call, meaning further functions calls will not be possible until it has completed. **is_initialised()** confirms the successful prior completion of the initialisation procedure. **start_sampling()** and **stop_sampling()** begin and halt the explicit sampling process. Not all ecl_daq methods have a meaningful implementation of this, but in some cases it may be advisable to only run the sampling process when it is needed. **is_sampling()** must be *TRUE* in order for online eye movement analysis to be performed in ecl_process. **get_samples()** is ecl_daq's central function, and is typically called by the ecl_process object. The timestamp it requires is taken straight from the ecl_process.fetch_and_process() function call, that is executed by the user application. However, it will only be used by buffered ecl_daq methods. These methods allow access to outdated eye position information; for instance, when the experimental application cannot keep up with the ecl_daq sampling rate while the user values temporal resolution over the real-time availability of information. Digitized versions of the DPI's four analog channel values are returned as a *long* between 0 and 4095. **monitor()** is a debugging function that allows the continuous output of raw data values to an ecl_monitor until an ecl_response is pressed. Note that this function is synchronous: invoking it will temporarily block the main application.

On a technical note, for readers unfamiliar with object-oriented programming, the *virtual* keyword preceding the return type signals that child classes can implement their own functionality for the base class function call. The *=0* code behind the function argument list indicates that any child class will need to provide its own implementation if the function call is to work at al. For *=0* functions, we provide no implementation in the base class itself.

```
bool start_recording(const char * fname, long nr_samples);
bool stop_recording();
```

Thus, these two functions are implemented in the ecl_daq base class itself. They provide the user with the opportunity to dump raw data readings to a text file, mainly to assist in debugging either hardware or application errors. For instance, raw eye movement data could be stored once, and the user could replay that exact same eye movement as if it were live data during the development of his software application, without needing to have a real subject in the DPI tracker. **start_recording()** requires a name for the dump file, to be created in the project’s main directory, and a maximal number of samples for which memory should be pre-allocated. **stop_recording()** ends the creation of the dump file. When recording is active, get_samples() calls will automatically add each new raw sample to the dump file. To replay the data contained in a dump file, use the ecl_daq_recording child class described below to feed data into ecl_process.

Child classes

Five implementations of the abstract ecl_daq methods are provided in the standard ECL libraries. Unless mentioned differently, no arguments need to be provided for construction or initialisation.

ecl_daq_ni makes use of a National Instruments BNC-2111 connector block and PCI-6221 analog-to-digital conversion card. This is an unbuffered method with effective millisecond-precise measurement. Since the hardware is in principle able of buffering its samples, a buffered method might be developed in the future. H, V, B, and T cables are to be connected to ports ai0, ai1, ai2, and ai3 respectively.

ecl_daq_visage uses the analog-to-digital conversion capabilities of the CRS Visage stimulus generator. H, V, B, T are to be connected to analog ports 1-4 on the Visage box. This is an unbuffered implementation relying on the GET_RESOURCE_VALUE call in the Visage's RTS language. When response buttons or monitors are connected to the Visage as well, ecl_daq_visage doubles as an input argument to ecl_response_visage and ecl_monitor_visage, respectively. Note, however, that due to the architecture of the RTS language, the actual temporal resolution of this method is capped at the refresh rate of the monitor. This makes it unsuitable for using the online saccade and fixation detection algorithms provided in ecl_process, which need an actual 1 ms temporal resolution.

ecl_daq_visage_w_sacc is an extended version using a hardware implementation of the saccade detection algorithms to solve this problem. The saccade detector is connected to the parallel port of the visage. In practice, the temporal resolution of pixel coordinates will still be capped at the monitor refresh rate, as will the read-in of the saccade detector output device, but the input signals to the saccade detection algorithm implemented within it will not be. Eye status information can then be accessed directly through ecl_daq_visage_w_sacc member functions, instead of through ecl_process (where detect_saccades() should be set to *FALSE*).

ecl_daq_mouse uses the Microsoft DirectX libraries to retrieve unbuffered samples from the computer mouse. Since a mouse cannot blink nor lose track, it is always set to be unblinked and tracking. This data acquisition method is mainly useful for debugging, simulation or illustration purposes on a computer that is not attached to a DPI tracker. The saccade detection algorithms will not be triggered by the comparatively slow mouse movements, but can be simulated by pressing the left mouse button. ecl_daq_mouse does not need to be calibrated when using the default ecl_config to ecl_process, as it returns straight pixel coordinates.

ecl_daq_recording enables the user to read and replay dump files created by the ecl_daq base class recording functions as if they were online data. A filename is required as an argument to the constructor. The first timestamp provided to get_samples() is used as a reference point that is matched with the first timestamp in the recorded file; subsequent timestamps will be regarded as relative to this reference point. After the recorded file has been replayed completely, the data acquisition device will appear to have lost track. Use stop_sampling() and then start_sampling() to restart the replay. Calibration of this DAQ method will have to be done through manually setting, or saving and loading, an appropriate ecl_config.

ecl_process

Description

ecl_process constitutes the core of ECL. Its main function is to convert the raw data samples delivered by ecl_daq into XY coordinates on the experimental monitor, as well as decide the current status of the eye. It executes this conversion every time the application uses its fetch_and_process() call – typically once every millisecond –
but it needs to have successfully completed a calibrate() routine first to establish the exact correspondence between raw data values and screen coordinates. By default, ecl_process will assume the ecl_daq device to return straight pixel coordinates. A brief drift_correction() routine is available for performing small lateral corrections to the stored calibration values. In addition, users can define rectangular regions of interest, both to check against the current eye position and to draw a schematic overview of the subject’s behavior quickly onto (most often) a control monitor.

Though the ecl_process base class is fully implemented, any of its functions can be overridden by a child class should the user wish to adapt it without modifying the existing code.

Base class definition

```
                ecl_process();
                ecl_process(ecl_config cfg);
virtual        ~ecl_eyetracker();

virtual bool init(ecl_daq * daq, ecl_monitor * subj_monitor,
                                          ecl_monitor * ctrl_monitor=NULL)
virtual bool is_valid();
virtual bool is_initialised();
```

For the object's construction, the user has a choice to specify his own ecl_config configuration, or use the default values described below in the ecl_config section. **init()** is the initialisation routine that needs to be called before using ecl_process further. It requires the specification of a data acquisition object, and the ecl_monitor viewed by the subject. A control monitor can be optionally provided. **is_valid()** returns the initialisation status of the ecl_daq and ecl_monitor objects, whereas **is_initialised()** returns the initialisation status of ecl_process itself.

```
virtual bool calibrate(ecl_response * rp);
virtual bool drift_correction(ecl_response * rp, int pix_x, int pix_y);
          void get_config(ecl_config & cfg);
```

calibrate() is the main calibration routine. It makes use of the subject and control monitors specified during the initialisation of ecl_process, but requires a direct ecl_response* argument to specify the response button to monitor. The subject is first presented with a central dot, allowing the experimenter to set up the DPI machine during central fixation. Calibration points are then shown one by one on the experimental monitor, in a diagonal cross shape going from the upper-left to the lower-right corner, and then from the lower-left to the upper-right corner. The subject is instructed to fixate each point as it appears. When a button is pressed, either by the subject or the experimenter, a number of samples before and after the button press, specified in ecl_config, are averaged to obtain the raw data measurement for this calibration point. After recording all calibration points, a linear fit is applied to the horizontal and the vertical raw data separately, and the calibration result is stored. If a control monitor was specified, the experimenter can view the eye position during calibration as well as the resulting goodness-of-fit. Should the precision of the calibration not suffice, the routine can be run again. The application can now retrieve the eye position in subject monitor pixel coordinates rather than digitized voltage coordinates. **drift_correction()** allows immediate adjustment of the constant part of the linear fits using only one calibration point. The user has to specify the ecl_response to monitor and the relevant pixel position on the subject screen, but is himself responsible for letting his application actually draw the calibration point on the screen. For instance, this could simply be the fixation cross shown at the start of a trial. Note though, that drift correction is only permitted within a limited range specified in ecl_config. More serious deviations from the previously established calibration will need to be corrected through a full calibrate() routine. **get_config()** extracts the full current configuration including calibration values.

```
virtual bool fetch_and_process(long tstamp=0);
```

fetch_and_process() is the central function of ecl_process, and performs two separate actions. First, it fetches raw data samples from the ecl_daq object provided to ecl_process. Then it processes these raw samples according to the present calibration values and configuration settings. A timestamp can be provided, indicating the current time in milliseconds. The ecl_get_time() auxiliary function described below can be used to retrieve this value. In case of buffered methods, it may sometimes be desirable to simply increment the timestamp with each function call regardless of the current time. If no timestamp is provided, the latest available sample will be used. The results of the fetch_and_processl() call are stored inside ecl_process, and can be retrieved using various functions.

```
virtual bool get_pix_position(long &x, long &y);
virtual bool get_daq_position(long &h, long &v);
```

get_pix_position() returns the current eye position in pixel coordinates, and should always be preceded by a fetch_and_process() call. Users may or may not want to call this function constantly, depending on whether their scientific needs extend beyond standard functions provided which automatically incorporate the current eye position, such as on_region(). **get_daq_position()** returns the current raw data values from which the pixel values were computed.

```
virtual void detect_saccades(bool onoff)
virtual bool is_in_oscillation();
virtual bool is_in_saccade();
virtual bool is_in_good_saccade();
virtual bool is_in_good_fixation();
virtual bool is_in_safe_saccade();
virtual bool is_in_safe_fixation();
virtual bool is_in_flock_saccade();
virtual bool is_in_flock_fixation();
virtual bool is_in_blink();
virtual bool is_in_track();
virtual bool is_in_timeout();
```

These functions retrieve specific aspects of the current eye state. Specific parameters for determining the eye state are defined in ecl_config. When **detect_saccades()** is set to *FALSE* the saccade detection algorithm is not applied during fetch_and_process() calls.
A **saccade** is detected using the algorithm described by Van Diepen (1998), on the basis of a five-element cyclical buffer storing the latest eye positions obtained. A **safe saccade** requires that two additional criteria are met: a stable velocity over the entire cyclical buffer, and a minimum duration during which is_in_saccade() was *TRUE*. In

general, if quick saccade detection is important, use is_in_saccade(); if not use is_in_safe_saccade() as a criterion.

An **oscillation** is the absence of a saccade when ecl_process is not in blink, loss of track or time-out, but no **safe fixation** is detected either. Safe fixations are detected when the deviation across the five-sample cyclical buffer does not exceed a given criterion value. A **false lock** occurs when the eye position is outside the calibrated area, augmented with a tolerance value. While false lock measurements are in principle valid, they are problematic because they determine pixel coordinates on the basis of extrapolation, rather than interpolation, from the calibration measurements. The opposite of a false lock is a **good** saccade, fixation or oscillation.

Blink and **track** are direct conversions of the analog DPI signals. If the subject is blinking, or the DPI machine has lost track, no further eye state can be determined. **Time-out** occurs when the latency between two successive samples is larger than the time-out limit. Typically, this means that the PC is being overburdened with calculations, either within the application or in background processes. Common causes include random network traffic or performing demanding graphical operations during eye tracking.

ecl_config

ecl_config is a class in global scope bundling the configuration parameters of ecl_process, as well as implementing save and load functionality to allow for reuse of the exact same configuration across different sessions. If no ecl_config is explicitly provided to ecl_process, default parameters defined within ecl_process will be used.

```
int num_cal_points;
int num_cal_samples_pre;
int num_cal_samples_post;
int deg_calrange_x;
int deg_calrange_y;
int deg_max_drift_x;
int deg_max_drift_y;
```

These parameters relate to the calibration routine. Respectively, they denote the number of calibration points measured (default: 10), the number of eye position samples collected before and after the button press during calibration (default: 150 and 0), and the calibration range in visual degrees for both directions (default: 15 and 10). The remaining two parameters define up to what error in visual degrees the drift correction routine is allowed to execute (default: 2 and 2).

```
bool           detect_saccades;
short int sac_lower_bound;
short int sac_upper_bound;
short int sac_safe_velocity;
short int sac_safe_duration;
```

These parameters relate to the saccade detection algorithm. detect_saccades allows the user to either enable or disable the algorithm from being executed at every ecl_process.fetch_and_process() call (default: on). The lower and upper bounds are used to quickly decide, based on the acceleration pattern, what is a saccade and what is not (default: 4 and 18). The remaining two parameters allow the detection of a safe saccade, when it has reached a certain stable velocity and duration (default: 50 deg/s and 5 ms).

```
double deg_fix_safe;
double deg_flocktol_x;
double deg_flocktol_y;
```

These parameters relate to the remaining decisions regarding the state of the eye. m_deg_fix_safe defines the maximal deviation of the eye to still be called a safe fixation (default: 0.5 degree), the latter two relate to the false lock tolerance around the calibrated zone (default: 1 and 1 degree).

```
double region_tol_x;
double region_tol_y;
```

ecl_process allows the definition of regions of interest on the screen. The region_tol_x and region_tol_y parameters define the extent of the tolerance zone used in determining whether the eye position is located inside such a zone (default: 1 and 1 visual degree).

```
short unsigned timeout_limit;
```

This parameter determines the maximal latency between the acquisition of two subsequent samples before a time-out occurs in ecl_process (default: 1 ms).

```
short int cal_vals_x[], cal_vals_y[];
short int cal_vals_h[], cal_vals_v[];
double      x_corr, y_corr;
double      x_constant, x_weight, y_constant, y_weight;
```

These variables store a specific ecl_process calibration. cal_vals_x and cal_vals_y contain the pixel coordinates at which calibration points were shown, whereas cal_vals_h and cal_vals_v contain the corresponding raw data values obtained at these coordinates. x_corr and y_corr are measures of goodness of fit, correlating pixel coordinates and raw data values. The remaining variables store the linear regression parameters that ecl_process will use to convert raw data into pixel coordinates. When manually setting up a calibration in ecl_config, only these regression parameters need to be provided for ecl_process to function.

```
virtual bool save_config(const char * fname);
virtual bool load_config(const char * fname);
```

Finally, the save and load functions of ecl_config allow these values to be stored to a file on the hard disk for retrieval in the future. This is only required when the intent is to easily *reproduce* the exact same configuration in the future; for logging purposes, configuration values are also stored in ecl_output files.

Regions

```
class ecl_process_region
{
public:
                ecl_process_region(int x, int y, int w, int h);
virtual         ~ecl_process_region();

        void draw(ecl_monitor * mon, ecl_LUT_color c=0);
        bool hit(int x, int y);
        void set_colors(ecl_LUT_color in, ecl_LUT_color out);
        void enable_automatic_drawing(bool onoroff);
};
```

ecl_process_region is a subclass of ecl_process, used to handle rectangular regions of interest on the screen defined by the user. During construction, its location and size can be specified in pixel coordinates; once created, it can be drawn onto an ecl_monitor using **draw()**. If a given coordinate is inside its region, **hit()** will return *TRUE*. **set_colors()** allows the modification of the default rendering colors of regions on ecl_monitors (grey when the eye is outside the region, white when it is inside). **enable_automatic_drawing()** allows the user to toggle on or off the automatic drawing of a specific region using the draw_ctrl_display() function described below. If set to *FALSE*, the region must be drawn individually using its draw() function.

Let us go back to the ecl_process base class now.

```
        int add_rectangular_region(int x, int y, int w, int h);
        ecl_process_region * get_region(int handle);
```

```
            int find_current_region();
            bool on_region(int handle);
            bool clear_regions();
```

Typically, new regions are created using ecl_process's **add_rectangular_region()** function, as this automatically stores their properties inside the ecl_process object. An integer handle is returned, which can, if necessary, be used to retrieve a pointer to the actual ecl_process_region object using **get_region()**. Handle values increase from 1 onwards as regions are added. **on_region()** compares the current eye position of ecl_process to the specified region, and returns *TRUE* if the eye is positioned inside the region (augmented with its tolerance zone). **find_current_region()** returns the handle of the numerically first region inside which the eye is currently located. **clear_regions()** releases the allocated memory for all regions, and resets the handle counter.

```
virtual bool draw_eye(ecl_monitor * mon, int size, ecl_LUT_color c=0);
virtual bool draw_ctrl_display();
```

Two additional drawing functions related to the current eye position are provided in the ecl_process base class. **draw_eye()** displays the current eye position on an ecl_monitor as a circular marker of specified color and size; if no color is provided, the monitor's default foreground color is used. **draw_ctrl_display()** is the standard shorthand function to generate an informative control display, combining the draw_eye() function with a ecl_process_region.draw() call for all regions declared using ecl_process.add_rectangular_region() - unless automatic drawing was turned off for an individual region. A standard color scheme is used, that can however be customized for individual regions through calling set_colors on the ecl_process_region pointer connected to the relevant handle. Importantly, all of these drawing functions require an ecl_monitor.present() call before becoming effective. We will discuss this in more detail in the ecl_monitor section below.

```
virtual bool monitor_text(ecl_monitor * mon, ecl_response * rp);
virtual bool monitor_visual(ecl_monitor * mon, ecl_response * rp,
                                    bool b_draw_regions=true);
```

Two standard ways of monitoring the operation of ecl_process are provided. **monitor_text()** renders the current pixel coordinates as text on an ecl_monitor until an ecl_response is pressed. **monitor_visual()** continuously draws the eye position on an ecl_monitor as a small circle, as well as, unless explicitly disabled, the regions of interest stored in ecl_process.

Output functionality

More often than not, researchers want to save their data. ecl_process implements functionality to automatically save current samples in a specific file format. Specifically, the user needs to allocate a buffer in memory before each trial, in which the samples will be stored. After the trial's end, the user is himself responsible for saving this buffer to a file on the hard disk, as well as clearing or destroying the buffer before the next trial. Note that for the correct calibration parameters to be applied to the saved data, the output buffer must be created *after* calibration or drift correction calls are made.

```
    bool output_create_buffer(ecl_output * format, int nsamples);
    bool output_clear_buffer();
    bool output_destroy_buffer();
    bool output_start_adding_samples();
    bool output_stop_adding_samples();
    bool output_add_comment(char * text);
    bool output_save_to_file();
```

output_create_buffer() requires an ecl_output object which implements both the handling of the temporary buffer and the saving of this buffer to a file. In addition, it requires the application to pre-allocate a maximal number of samples to be saved in the buffer. **output_clear_buffer()** purges all samples from the buffer, whereas

output_destroy_buffer() frees up the memory the buffer was occupying, necessitating a new output_create_buffer() call before more data can be saved. **output_start_adding_samples()** and **output_stop_adding_samples()** toggle the automatic saving of the current data samples during each ecl_process.fetch_and_process() call on or off. **output_add_comment()** saves any text comment at the current timestamp. Finally, **output_save_to_file()** writes the output buffer to a file in a format determined by the specific ecl_output child class that was provided to output_create_buffer().

ecl_output

Description

The ecl_output base class fully implements a buffering system to hold the incoming data samples, but leaves the actual writing of a file to its child classes. Currently, only the p-file format used in the old DPI software (Van Diepen, 1998) is available. Typically, users will not interact directly with an ecl_output object, but will pass it on to an ecl_process object and use the output functions implemented in that class. It is, however, possible to manually write files as well.

Base class definition

```
               ecl_output(const char * fname);
virtual       ~ecl_output();

          bool create_buffer(long numsamples, ecl_config cfg);
          bool clear_buffer();
          bool destroy_buffer();
          bool add_daq_sample(long tstamp, long h, long v, bool b, bool t,
                              short int state);
          bool add_comment(long timestamp, char * txt);

virtual bool save_buffer()=0;
```

create_buffer() sets aside memory for a maximal number of samples, and stores the relevant configuration. **clear_buffer()** purges all data from the buffer, while **destroy_buffer()** releases it entirely. In order to add a sample to the buffer, **add_daq_sample()** is called. Aside from a timestamp, it requires the raw ecl_daq values for eye position coordinates, booleans for blink and track, and a coded two-byte value for the current eye state. Currently only the lower byte is used (adapted from Van Diepen, 1998):

0	**Right key down**
1	**Left key down**
2	**Safe**
3	**Fixation**
4	**False Lock**
5	**Time-out**
6	**Track/Blink**
7	**Signal error**

add_comment() allows adding a text comment at the specified timestamp. **save_buffer()** is to be called once all samples for the current trial have been collected, and adds the contents of the entire buffer to the output file specified during construction. Note, again, that this specific functionality needs to be implemented by a child class. This allows easy definition of new file formats.

<u>Child classes</u>
ecl_output_p implements the existing p-file format for ECL (Van Diepen, 1998). A data viewer and offline analysis program are available on request. Due to the disk space constraints of the past, the p-file format is heavily compressed; with storage space now more abundant, we plan to develop a more easily readable file format in the future.

ecl_monitor
<u>Description</u>

ECL provides a standardized abstract base class to perform drawing operations onto monitors, allowing flexible switching between different monitor setups with minimal impact on existing code. For instance, a user could program his or her experiment on a desktop PC using MS DirectX to visualize experimental stimulation, and then easily port the experiment to a specialized Visage configuration when finished. However, some advanced functions might not be implemented in ecl_monitor. For these functions we recommend the user either extends the existing ecl_monitor child classes with the required functionality, or uses direct calls to the relevant API alongside its encapsulation in an ecl_monitor object.

<u>Base class definition</u>

```
                ecl_monitor();
virtual        ~ecl_monitor();

virtual bool    init(long w_mm, long h_mm, long d_mm, const char * name)=0;
virtual bool    is_initialised()=0;
virtual int     get_pix_width()=0;
virtual int     get_pix_height()=0;
virtual int     get_pix_depth()=0;
virtual int     get_framerate()=0;
virtual void    wait_for_vertical_blank()=0;
        double pix_to_deg(int pix_dist);
        int     deg_to_pix(double deg_dist);
```

init() is the initialisation routine. It requires the user to define the size of the screen surface and the distance to the screen in mm, for use in calculations dependent on measurements expressed in visual angle rather than pixels. In addition, a custom name is to be provided for use in output files. **is_initialised()** returns a boolean to indicate whether initialisation was performed succesfully. **get_pix_width()** and **get_pix_height()** are used to retrieve the current spatial resolution of the monitor. **get_pix_depth()** retrieves the pixel depth, in bits, and **get_framerate()** the temporal resolution in Hz. **wait_for_vertical_blank()** pauzes the application until the scan beam of the CRT monitor reaches the top of the screen. **pix_to_deg()** converts a given pixel distance on the screen into degrees of visual angle, using the properties of the ecl_monitor. **deg_to_pix()** performs the opposite operation, rounded to the nearest integer.

```
            void set_bg_color(ecl_LUT_color c);
            void set_fg_color(ecl_LUT_color c);

virtual void clear_screen()=0;
virtual void clear_rect(int x, int y, int w, int h)=0;
virtual void draw_bitmap(HBITMAP bmp, int x, int y, int w, int h)=0;
virtual void draw_text(int x, int y, char * txt, ecl_LUT_color c=0)=0;
virtual void set_draw_text_fontsize(int s)=0;
virtual void set_font(char * name)=0;
virtual void draw_line(int x1, int y1, int x2, int y2, int w, ecl_LUT_color
                                  c=0)=0;
virtual void draw_cross(int x, int y, int size, ecl_LUT_color c=0)=0;
virtual void frame_rectangle(int x, int y, int w, int h, ecl_LUT_color c=0)=0;
virtual void fill_rectangle(int x, int y, int w, int h, ecl_LUT_color c=0)=0;
virtual void frame_oval(int x, int y, int w, int h, ecl_LUT_color c=0)=0;
virtual void fill_oval(int x, int y, int w, int h, ecl_LUT_color c=0)=0;
virtual void draw_pixel(int x, int y, ecl_LUT_color c=0)=0;

virtual bool present()=0;
            void set_throttle(int thr);
```

This is the standard set of drawing functions for an ecl_monitor. **set_bg_color()** is a function implemented in the base class to uniformly set a background color for all drawing operations to this monitor (default: black). A separate ecl_LUT_color class is provided to handle colors (see below). **clear_screen()** and **clear_rect()** clear either the whole screen or a rectangular area defined in pixel coordinates to this background color. **set_fg_color()** sets an overall foreground color (default: white), that can however usually be overridden by individual drawing functions. The other drawing functions are self-explanatory.

present() is a very important function. Typically, drawing operations to ecl_monitors are performed offscreen, in video memory. This avoids many undesirable effects such as flickering, but does require an additional call - present() - to actually show the visual stimulation onto the monitor when all drawing has finished for a particular frame. Continuously changing graphics can be generated using repeated drawing and present() calls in a while-loop. However, this often places an unnecessary burden on the experimental computer, since a modern PC’s ability to run through these while-loops is far greater than the temporal resolution of visual perception or even the monitor itself. To alleviate this problem, **set_throttle()** can be used to automatically restrict these operations to a limited number of executions per second for a given monitor. By default, the throttle value is equal to the refresh rate, but especially in case of a control monitor, far less frequent updates may suffice. Note though that although set_throttle() is implemented in the base class, a child class needs to properly implement it in its present() and drawing calls.

```
void add_scrolling_text(const char * txt);
void clear_scrolling_text();
void draw_scrolling_text();
```

Using the text drawing functions of a child class, the ecl_monitor base class implements functionality to store and render a text buffer. Each **add_scrolling_text()** call adds a line of text to this buffer, purging the oldest text as the buffer exceeds the screen dimensions and shifting the entire text display one line up. **clear_scrolling_text()** clears the entire text buffer, and **draw_scrolling_text()** clears the current ecl_monitor to its background color, then draws the entire text buffer to the screen using the default foreground color and font. Remember though, that a present() call is still required for the drawing operations to take effect on the screen itself.

Child classes

Two standard implementations of ecl_monitor are provided.

ecl_monitor_directX implements the above methods for Microsoft DirectX 9. DirectX should be the standard choice for controlling the control monitor, but can also be used for experimental stimulation. Advanced functions are available in the ecl_monitor_directX class to encapsulate more complex DirectX functionality in easy-to-use functions specifically for this purpose. Most prominently, whereas the base class function implementations at the same time generate DirectX surfaces and blit them to video memory, these advanced functions allow the preloading of DirectX surfaces to avoid interference of complex surface generation (such as text or bitmaps) with the temporal precision of eye movement data sampling. The ecl_monitor_directX initialisation routine is adapted from its base class declaration and requires the specification of the monitor number, as displayed in the Windows display manager, the instance number of the DirectX window to be used, the physical size of and distance to the monitor, and a custom name for this monitor.

ecl_monitor_visage provides methods for writing to a screen controlled by a CRS Visage stimulus generator. For advanced experimental stimulation, a specific professional solution such as the Visage is preferable over DirectX or other general stimulation methods. All normal Visage functionality, including RTS scripting, is still available in the global scope, and should preferentially be used for complex or time-critical operations. In its initialization routine, ecl_monitor_visage uses the standard base class arguments as well as a pointer to the ecl_daq_visage object. An ecl_daq_visage should always be provided, even when eye movement signals are not collected through the Visage. In that case, two different ecl_daq objects might exist.

ecl_LUT_color

The reader will have noticed that color values are expressed as ecl_LUT_color objects. At the moment, this is just a typedef for an integer, to which color values can be assigned using the RGB macro defined in *windows.h*. In the future, the ecl_LUT_color system could be expanded to properly handle LUTs.

ecl_response

Description

Analogous to ecl_monitor, ecl_response allows the standardisation of various methods of manual response collection for use with ECL. At the moment only binary responses are implemented, but in the future more complex response devices may be added.

Base class definition

```
                ecl_response();
virtual        ~ecl_response();

virtual bool is_pressed()=0;
virtual void wait_until_pressed();
virtual void wait_until_released();
virtual void wait_until_pressed_then_released();
```

is_pressed() is the only method any child class has to implement; the others derive from it, but can also have a custom definition inside a child class. Logically enough, it returns the current state of the response device as a boolean. In case of a continuous measurement device, a threshold could be set to convert it into a boolean so that it can be used with other ECL classes.

```
virtual bool monitor(ecl_monitor * mon);
```

monitor() displays the current ecl_response status to an ecl_monitor until the Escape key is pressed on the keyboard.

Child classes

ecl_response_keyboard takes the relevant keycode as a parameter to its constructor, and uses the standard Windows API to access the keyboard. Note that keyboards are not suitable for producing accurate reaction time measurements.

ecl_response_mouse uses the DirectX mouse architecture for accessing its buttons. As an argument to its constructor, it takes the number of the relevant mouse button.

ecl_response_parallel_port uses ecl_parallel_port to decide whether a response device connected to the PC's own parallel port is pressed or not. The constructor takes as an argument the pin to which the response device is connected. Please read the ecl_parallel port section for detailed instructions.

ecl_response_visage is similar to ecl_response_parallel_port, but uses the Visage's parallel port. Again the constructor requires a pointer to ecl_daq_visage as an additional argument. Note that, as with eye movement sample collection, the temporal resolution of response collection is capped at the refresh rate of the monitor.

ecl_parallel_port

This class (in global scope) is a wrapper around inpout32.dll, that allows the user to separately access pins on his PC's parallel port under Windows 2000, XP and Vista. This DLL needs to be present in the directory of the compiled executable in order to use the PC's parallel port for collecting responses.

```
bool   init(short port_address, bool set_input_mode);
short get_byte_value();
void   set_byte_value (short v);
bool   get_bit_value(short bitnum);
void   set_bit_value(short bitnum, bool v);

void   monitor(ecl_monitor * mon, ecl_response * rp);
```

The **monitor()** functions again allows direct monitoring of the parallel port status until an ecl_response device is activated.

Auxiliary functions

Finally, ECL provides some functions in the global scope that are used throughout the ECL libraries, but can also be useful for implementing experimental designs in C++.

```
 bool ecl_die(char * error);
 bool ecl_key_down(int key_code);
 void ecl_wait_for_key_press(int key_code);
 long ecl_get_time();
long ecl_reset_timer();
void ecl_wait_time(long msec);
```

ecl_die() throws a Windows exception in case of an insurmountable error. **ecl_key_down()** and **ecl_wait_for_key_press()** enable the user to use keyboard keys in his program without the need to define a full-fledged ecl_response. **ecl_get_time()** uses the Windows function QueryPerformanceCounter() to retrieve the current time, in milliseconds. **ecl_reset_timer()** resets this timer to 0, whereas **ecl_wait()** uses ecl_get_time() to halt *all* operation until a given time has passed.

Conclusions

We have presented an updated library of C++ functions to employ the classic Dual Purkinje Image eye tracker in modern vision science. ECL was designed to be modular and easily extended by third party programmers, as well as thoroughly complete in the availability of useful routines and functions for eye tracking research. For more information, we refer to the ECL website:

http://ppw.kuleuven.be/labexppsy/ECL/