

Eye-tracking API interface/functions: State-Of-The-Art

Let's observe some existing interfaces (or set of functions in case of low-level libraries) that are used now to communicate with devices.

LG EyeGaze

The API consists of 3 main functions:

```
int EgInit(struct _stEgControl *pstEgControl);

int EgExit(struct _stEgControl *pstEgControl);

void EgCalibrate(struct _stEgControl *pstEgControl, HWND
    hwnd, int iCalAppType);
    stEgControl Control and status variables used to setup and control the Eyegaze
    image processing software. hwnd window handle for the application program
    window iCalAppType EG_CALIBRATE_DISABILITY_APP = 0 or
    EG_CALIBRATE_NONDISABILITY_APP = 1

int EgGetData(struct _stEgControl *pstEgControl);
    /* Retrieve data collected by eyegaze image processing thread */
    /* EgGetData() returns the buffer index of the last gaze point sample */
    /* measured by Eyegaze */

struct _stEgControl
{
    /* CONTROL INPUTS FROM APPLICATION: */

    struct _stEgData *pstEgData; /* pointer to the Eyegaze data structure */
    /* where EgGetData() places the next gaze point data sample. */
    /* The application must set this buffer address in the stEgControl */
    /* structure before calling Eglinit() */

    int iNDataSetInRingBuffer; /* number of gaze point data samples in */
    /* Eyegaze's internal ring buffer. The application must set the ring- */
    /* buffer length in the stEgControl structure before calling Eglinit() */

    BOOL bTrackingActive; /* flag controls whether eyetracking is */
    /* presently on (TRUE = 1) or off (FALSE = 0). If the flag is on when */
    /* a new camera field finishes, the Eyegaze thread processes the image */
    /* and puts the results in the data ring buffer; if the flag is off, */
    /* the camera field is not processed. The application may turn this */
    /* tracking flag on or off at any time. NOTE for non-Windows users: BOOL type */
    /* is a 4-byte integer */

    int iScreenWidthPix; /* Pixel dimensions of the full */
}
```

```

int iScreenHeightPix; /* computer screen */

BOOL bEgCameraDisplayActive; /* flag controls whether or not the */
/* full 640x480 image from the Eyegaze camera is displayed in a separate */
/* window on the VGA. The application must set this flag */
/* prior to calling EglNit() and may turn the display flag on and off */
/* at any time. */

int iEyeImagesScreenPos; /* Screen position of the eye images. */
/* 0 = upper left, 1 = upper right */

int iCommType; /* Communication type: Comp Config: */
/* EG_COMM_TYPE_LOCAL, Single */
/* EG_COMM_TYPE_SOCKET, or Double */
/* EG_COMM_TYPE_SERIAL. Double */

wchar_t *pszCommName; /* Pointer to serial port name or IP */
/* address of server machine. used only in Double Computer Config */
/* Note that this is a "wide character" */

int iVisionSelect; /* Reserved - set to 0 (unused) */

/* OUTPUTS TO APPLICATION: */

int iNPointsAvailable; /* number of gaze point data samples */
/* presently available for the application to retrieve from Eyegaze's */
/* internal ring buffer */

int iNBufferOverflow; /* number of irretrievably missed gaze- */
/* point data samples, i.e. the number of valid data points at the tail of */
/* the ring buffer that the application did not retrieve and that Eyegaze */
/* overwrote since the application last called EgGetData(). */

int iSamplePerSec; /* Eyegaze image processing rate - */
/* depends on the camera field rate: */
/* RS_170 60 Hz */
/* CCIR 50 Hz */

float fHorzPixPerMm; /* Eyegaze monitor scale factors */

float fVertPixPerMm; /* (pixel / millimeter) */

void *pvEgVideoBufferAddress; /* address of the video buffer containing */
/* the most recently processed camera image field */

```

```

        /* INTERNAL EYEGAZE VARIABLE          */

void    *hEyegaze;    /* Eyegaze handle -- used internally by */
    /* Eyegaze to keep track of which vision subsystem is in use. */
    /* (not used by application)          */
};

struct _stEgData
{
    BOOL    bGazeVectorFound;    /* flag indicating whether the image */
    /* processing software found the eye, i.e. found a valid glint pupil vector */
    /* (TRUE = 1, FALSE = 0). NOTE for non-Windows users: BOOL type */
    /* is a 4-byte integer          */

    int    iIGaze;    /* integer coordinates of the user */

    int    iJGaze;    /* gazepoint referenced to the full computer screen (pixels) */
    /* 0,0 origin at upper left corner iIGaze positive iJGaze positive downward */

    float    fPupilRadiusMm;    /* actual pupil radius (mm) */

    float    fXEyeballOffsetMm; /* offset of the eyeball center from */

    float    fYEyeballOffsetMm; /* the camera axis (mm) */
    /* Notes on polarity: x positive: head moves to user's right; */
    /* y positive: head moves up */

    float    fFocusRangeImageTime; /* distance from the camera sensor plane */
    /* to the camera focus plane, at the time the camera captured the image (mm) */

    float    fFocusRangeOffsetMm; /* range offset between the camera focus */
    /* plane and the corneal surface of the eye - at image time (mm) */
    /* A positive offset means the eye is beyond the lens' focus range. */

    float    fLensExtOffsetMm; /* distance that the lens extension would */
    /* have to be changed to bring the eye into clear focus (mm) (at image time) */

    ULONG    ulCameraFieldCount; /* number of camera fields, i.e. 60ths of */
    /* a second, that have occurred since the starting reference time (midnight */
    /* January 1, this year) */

    double    dGazeTimeSec;    /* The application time that the gazepoint */
    /* was actually valid. (dGazeTimeSec represents the original image-capture */
    /* time, not the time that the gazepoint calculation was completed.) */
};

```

```
double dAppMarkTimeSec; /* Pentium TSC counter value at the moment */
/* that the mark was incremented. */

int iAppMarkCount; /* Mark count used in logging functions. */

double dReportTimeSec; /* The application time that Eyegaze */
/* reported the gaze point */
};
```

SR EyeLink

The API consists of many functions and is very complex; the minimal (?) set contains the following functions:

```
INT16 open_eyelink_connection(INT16 dummy);
    // Set up the EyeLink system and connect to tracker
    // If <dummy> not zero, will create a dummy connection
    // where eyelink_is_connected() will return -1
    // Returns: 0 if success, else error code

VOID close_eyelink_connection(void);

INT16 do_tracker_setup(void);
    Calibration. Also uses get_display_information(), init_expt_graphics(),
    set_offline_mode() and close_expt_graphic() functions as support of graphics of
    calibration window

INT16 start_recording(INT16 file_samples, INT16
    file_events, INT16 link_samples, INT16 link_events);
    /* Arguments indicate whether samples event are streaming */
    /* to a file and client application */

VOID stop_recording(void);

INT16 WINAPI eyelink_get_next_data(void FARTYPE *buf);
    /* makes copy of next queue item (ALLF_DATA) */
    /* if <buff> is NULL, just gets type */
    /* returns item type: */
    /* SAMPLE_TYPE if sample, 0 if none, else event code */

INT16 WINAPI eyelink_get_last_data(void FARTYPE *buf);
    /* makes copy of last item (ALLF_DATA) from eyelink_get_next_data */
    /* returns item type: */
    /* SAMPLE_TYPE if sample, 0 if none, else event code */

typedef union {
    FEVENT    fe;
    IMESSAGE  im;
    IOEVENT   io;
    FSAMPLE   fs;
    } ALLF_DATA ;

typedef struct {
    UINT32 time;    /* time of sample */
```

```

    INT16  type; /* always SAMPLE_TYPE */
    UINT16 flags; /* flags to indicate contents */
    float  px[2], py[2]; /* pupil xy */
    float  hx[2], hy[2]; /* headref xy */
    float  pa[2]; /* pupil size or area */
    float  gx[2], gy[2]; /* screen gaze xy */
    float  rx, ry; /* screen pixels per degree */
    UINT16 status; /* tracker status flags */
    UINT16 input; /* extra (input word) */
    UINT16 buttons; /* button state & changes */
    INT16  htype; /* head-tracker data type (0=none) */
    INT16  hdata[8]; /* head-tracker data (not prescaled) */
} FSAMPLE;

typedef struct {
    UINT32 time; /* effective time of event */
    INT16  type; /* event type */
    UINT16 read; /* flags which items were included */
    INT16  eye; /* eye: 0=left,1=right */
    UINT32 sttime, entime; /* start, end times */
    float  hstx, hsty; /* starting points */
    float  gstx, gsty; /* starting points */
    float  sta;
    float  henx, heny; /* ending points */
    float  genx, geny; /* ending points */
    float  ena;
    float  havx, havy; /* averages */
    float  gavx, gavy; /* averages */
    float  ava;
    float  avel; /* avg velocity accum */
    float  pvel; /* peak velocity accum */
    float  svel, evel; /* start, end velocity */
    float  supd_x, eupd_x; /* start, end units-per-degree */
    float  supd_y, eupd_y; /* start, end units-per-degree */
    UINT16 status; /* error, warning flags */
} FEVENT;

typedef struct {
    UINT32 time; /* time message logged */
    INT16  type; /* event type: usually MESSAGEEVENT */
    UINT16 length; /* length of message */
    byte  text[260]; /* message contents (max length 255) */
} IMESSAGE;

typedef struct {
    UINT32 time; /* time logged */

```

```
INT16  type;          /* event type: */
UINT16 data;         /* coded event data */
} IOEVENT;
```

SMI iViewX

There is no low-level library with API for this device. Rather, COM interfaces provide all necessary services. The obvious advantage here is that it is possible to calibrate the device using a single function DoCalibrate(ULONG points, LONG backColor, EyeType eye) of ICalibrate interface that is referred as *internal* calibration. The external calibration mean implementation of own calibration routine using the rest of functions of this interface (aka Tobii API)

```
VARIANT_BOOL Initialize ( VARIANT_BOOL demo )
    /* Prepares the eye tracker module. Establishes a socket connection to the iViewX
    application.
    Arguments
    - demo: Set to FALSE for now. Future releases will start the eye tracker in a demo
    mode which generates faked data. The demo mode will allow to test the API
    without using a real iViewX eye tracking system. */
```

```
VARIANT_BOOL Exit ( void )
    /* Cleans up memory. Closes socket connection to the iViewX application. */
```

```
VARIANT_BOOL SetBindIP ( LPWSTR ipAddress, ULONG port )
    /* This method must be called before Initialize(). It sets the listen IP address and
    port for the udp communication socket. This address must match with the target
    address of the iViewX application. Make sure that no firewall is blocking the port.
    Arguments
    - ipAddress: String for the listen IP address. i.e. "127.0.0.1" for the loopback
    interface
    - port: The listen port number. You should not use a well known port number
    between 0 and 1024.*/
```

```
VARIANT_BOOL SetTargetIP ( LPWSTR ipAddress, ULONG port )
    /* This method must be called before Initialize(). It sets the target IP address and
    port for the udp communication socket. This address must match with the listen
    address of the iViewX application. Make sure that no firewall is blocking the port.
    Arguments
    - ipAddress: String for the target IP address. i.e. "127.0.0.1" for the loopback
    interface
    - port: The listen port number. You should not use a well known port number
    between 0 and 1024 */
```

```
VARIANT_BOOL StartEyeData( void )
    /* Instructs the eye tracker to provide eye data. Call this before polling the eye
    tracker with GetEyeData */
```

```
VARIANT_BOOL StopEyeData( void )
    /* Instructs the eye tracker to stop sending eye data. */
```

VARIANT_BOOL GetEyeData(EyeData* left, EyeData* right)
/* This call doesn't set a "last error" if it returns false. It returns false if there are no more new EyeData samples available. It returns true if there are some more samples available. If your eye tracker generates samples at a rate of 200Hz and you poll the eye tracker every 100 ms for data. You can do 20 calls to GetEyeData before it returns false, because the eye tracker generates 20 samples in 100 ms. GetEyeData always returns the oldest sample. If you want to get only the newest one you have to read data as long as GetEyeData returns true and use the last read sample. If your eye tracker provides just left eye data you can pass a Null reference as right EyeData parameter. In a binocular system you pass both parameter.

Arguments

- *left*: Reference to an EyeData object receiving the left eye data.

- *right*: Reference to an EyeData object receiving the right eye data.

An EyeData object contains the timestamp, the gaze x/y, the pupil diameter x/y, pupil position x/y and a frame counter of the scene video if the eye tracker uses a scene video. */

VARIANT_BOOL IsEyeData(ULONG timeout)

/* This call blocks the caller until an eye data sample is available. When IsEyeData returns true you can fetch the sample with GetEyeData. IsEyeData returns false when is given timeout is exceeded and no sample was received. In a 50Hz eye tracker IsEyeData will release the caller every 20 ms.

Arguments

- *timeout*: Timeout in ms before the blocking function returns with false. */

CALIBRATION

VARIANT_BOOL DoCalibration(ULONG points, LONG COLOR, EyeType eye)

/* Performs a subject calibration. A sequence of screen points is shows to the subject on a full screen calibration window. The subject has to fixate the points. From the screen points and the fixation data a mapping functions is determined to transform any pupil position in the eye video into gaze data. This functions blocks the caller until the calibration is interrupted or finished successfully. The subject can accept points manually when pressing the space key or break the calibration by pressing the esc key. Gaze data is only available after a successful calibration using the GetEyeData method. DoCalibration performs an internal calibration. The calibration is done on the pc where the com server is running.

Arguments

- *points*: Number of points to specify the calibration method. Valid value are 1,2,5,9,13.

- *color*: Determines the background color of the full screen calibration window. This color should be similar to the color of the stimulus images to prevent a pupil center shift between calibration and experiment. PCS occurs when the pupil diameter changes (due to brightness) during an experiment leading to inaccurate gaze data. With Visual C++ or VB the color can be set using the RGB macro. With Visual C# the color can be set using the Color class.

- *eye*: Specifies which eye is to be calibrated in a binocular system. Possible options are EYE_BOTH, EYE_LEFT, EYE_RIGHT. In a monocular system you can safely always use EYE_BOTH.

VARIANT_BOOL StopCalibration(void)

Stops a running internal or external calibration and hides the full screen calibration window. No gaze data is available if a calibration was stopped.

VARIANT_BOOL StartCalibration(ULONG points, ULONG sizeX, ULONG sizeY, EyeType eye)
Starts an external calibration. Your application is responsible for showing the calibration points to the user. StartCalibration instructs the eye tracker to begin with the calibration procedure. Calibration is required to map raw eye position data into gaze data. During calibration a sequence of points is shown to the subject.

Arguments

- points 1,2,5,9,13 number of points specifying the calibration method
- sizeX horizontal size of the calibration area (resolution of the screen where the subject is looking)
- sizeY vertical size of the calibration area (resolution of the screen where the subject is looking)
- eye Specifies which eye is to be calibrated in a binocular system. Possible options are EYE_BOTH, EYE_LEFT, EYE_RIGHT. In a monocular system you can safely always use EYE_BOTH.

VARIANT_BOOL GetCurrentCalibrationPoint(LONG* point)

Returns the current calibration point when doing an external calibration. The current calibration point can be randomised or it can be a point out of a fixed sequence.

Arguments

- *point*: number specifying the current point, dependent on the calibration method *this* number ranges from 1 to 13

VARIANT_BOOL GetCalibrationPointPosition(LONG point, ULONG* posX, ULONG* posY)

Returns the position of the given calibration point when doing an external calibration. The point must be within the range allowed for the chosen calibration method. The position is given in screen pixels x,y.

Arguments

- *point*: number of the point where you want the position from
- *posX*: horizontal position of the point
- *posy*: vertical position of the point

VARIANT_BOOL AcceptPoint(void)

Accepts the current calibration point when doing an external calibration. If the subject looks at the current calibration point either the subject itself or the operator has to accept the point. After a successful acceptance of a point the next uncalibrated point will be shown until all points are accepted. In an auto calibration mode only the first point has to be accepted.

VARIANT_BOOL SetCalibrationPointPosition(LONG point, ULONG xPos, ULONG yPos)

Move a calibration point to the given position. Call this for every calibration point before an internal calibration. You can call SetCalibrationPointPosition at any time when doing an external calibration.

Arguments

- *point*: number of the point you want to move
- *posX*: new horizontal position of the point
- *posy*: new vertical position of the point

VARIANT_BOOL SetWaitValidData (VARIANT_BOOL on)
Enables the wait for valid data option of the calibration. When waiting for valid data a calibration point can only be manually accepted when the subject is fixating a screen point.
Arguments
- *on*: TRUE to enable, FALSE to disable the option

VARIANT_BOOL GetWaitValidData (VARIANT_BOOL* on)
Read if the wait for valid data option is enabled or disabled.
Arguments
- *on*: Reference to a variable receiving the state of the option

VARIANT_BOOL SetRandomizePoints (VARIANT_BOOL on)
Enables / Disables the randomise points option. When randomising points the calibration screen points are shown in an arbitrary order to the subject. The first point shown is always the center point. Randomise points is used for subjects which are used to the calibration procedure to prevent that the trained subject already knows where the next point will be. This would add a systematic error to the calibration.
Arguments
- *on*: TRUE to enable, FALSE to disable the option

VARIANT_BOOL GetRandomizePoints (VARIANT_BOOL* on)
Read if the randomise points option is enabled or disabled.
Arguments
- *on*: Reference to a variable receiving the state of the option

VARIANT_BOOL SetAutoAccept (VARIANT_BOOL on)
Enables / Disables the auto accept option. When auto accept is on the system will show the calibration screen points to the subject and move to the next point as soon as a fixation is detected. Every point will be calibrated and calibration will stop automatically in case of success. The auto accept function requires only manual acceptance of the first calibration point. The subject can accept the point by pressing the space key.
Arguments
- *on*: TRUE to enable, FALSE to disable the option

VARIANT_BOOL GetAutoAccept (VARIANT_BOOL* on)
Read if the auto accept option is enabled or disabled.
Arguments
- *on*: Reference to a variable receiving the state of the option

```
struct tagEyeData
{
    __int64 time;
    LPWSTR frameCounter;
    double gazeX;
    double gazeY;
    double diaX; /* diameter of pupil */
    double diaY;
```

```
double pupilX; /* pupil position */  
double pupilY;  
double reflexX;  
double reflexY;  
};
```

Tobii (all)

The API has only 4 main functions.

```
long __stdcall Tet_Connect(char *pServerAddress, long
    portnumber, ETet_SynchronizationMode syncmode);
    /* requires IP, port and synchronization type */

long __stdcall Tet_Disconnect(void);

long __stdcall Tet_Start(Tet_CallbackFunction Fnc, void
    *pApplicationData, long interval);
    /* stop the thread execution here until Tet_Stop is called. Fnc callback function is
    called with every gaze sample that becomes available. pApplication data is any
    value that is passed to the Fnc callback function. Interval is usually 0 (automatic).

long __stdcall Tet_Stop(void);
```

CALIBRATION

```
long __stdcall Tet_CalibLoadFromFile(char *pFile);
    Sets a calibration stored in a file. A file that has been stored by the
    Tet_CalibSaveToFile is read and written to the calibration in use and to the
    calibration under construction. Any old data is overwritten.
    Arguments:
    - pFile [IN]: Pointer to a null terminated array of chars that contains the path and
    filename to a calibration file created by the Tet_CalibSaveToFile call.
    Modifying:
    The calibration in use and the calibration under construction.

long __stdcall Tet_CalibSaveToFile(char *pFile);
    Saves the current calibration in use to file for a future reuse. Call
    Tet_CalibLoadFromFile at any time to load the calibration to the eye tracker.
    Arguments:
    - pFile [IN]: Pointer to a null terminated array of chars that contains the path and
    filename of the calibration file to store.
    Modifying:
    Creates a new calibration stored in a file. If the file exists, the old one is overwritten.

long __stdcall Tet_CalibClear(void);
    This call deletes all samples that are in the calibration under construction. It is a
    good practice to always start a new calibration with this function call.
    Modifying:
    The calibration under construction.
```

```
long __stdcall Tet_CalibAddPoint(float x, float y, long
nrofdata, Tet_CallbackFunction Fnc, void
*pApplicationData, long interval);
```

This is a blocking function that will start collecting gaze data samples for a specific gaze target point. The samples are added to the set of new data in the calibration under construction. Normal usage is to display something at a known position (x, y) and make sure subject gazes at this point. Then call this function to make the system try to record a specified number of samples. This procedure is repeated for each point to add to the calibration under construction. When points are chosen, spread them over the area that will be gazed at during eye tracking later. Have in mind that the more points the better. However, more than, say, 10 points will not significantly improve the quality. When all points are added, call Tet_CalibCalculateAndSet to copy all the calibration data from the calibration under construction to the calibration in use. After this a call to the Tet_CalibGetResult can be made to inspect the samples to see if there are some improvements to the samples that can be made. It is not possible to inspect the samples before Tet_CalibCalculateAndSet is called. For single threaded applications or applications that must be able to prematurely stop the adding of a calibration point, the blocking nature of Tet_CalibAddPoint may be unacceptable. Therefore, it is possible to arrange for a callback whenever there are either data available or a periodic timer event occurs. This feature is the same as for the Tet_Start. When new gaze data is available, the user defined callback function Fnc will be called immediately with fresh data and ETet_CallbackReason parameter set to TET_CALLBACK_GAZE_DATA. The pData parameter will point to a STet_GazeData struct. To make the callback function called at regular interval, even though there are no new gaze data, set the interval parameter to something else than zero and the callback function will periodically be called with ETet_CallbackReason TET_CALLBACK_TIMER and pData set to NULL. For instance, this makes it possible to write single threaded applications and still make the user interaction and GUI updates possible. There are some important considerations regarding the time consumed by the callback function. See the description of Tet_CallbackFunction for details about this issue. Of course, there's an option to not use the callback, which is the natural choice for many applications. Just let the call block and wait for it to finish or fail. Do this by passing NULL as the callback function Fnc. Tet_CalibAddPoint will exit when it is done, on unrecoverable error or if Tet_Stop prematurely stops the calibration. To stop, call Tet_Stop in your callback function. Note that a Tet_CalibAddPoint may be called in several threads. The Tet_Stop will match only the Tet_CalibAddPoint within the same thread. Note that nrofdata is a hint to the system of how many samples it will try to acquire. Never rely on this number in code. There may be much more callbacks if eye tracking conditions are poor and it may be less if an error occurs. Typically if nrofdata is set to 6 there will be 12-24 callbacks if no errors occur.

Arguments:

- x [IN]: Horizontal position of target point ranging from 0 to 1 where 0 is leftmost position and 1 is rightmost from eye tracking subject point of view. For example, if a monitor is the target, a good choice is to set the leftmost position to 0 and rightmost position to 1, i.e. the position unit will be percentage of a screen.
- y [IN]: Vertical position of target point ranging from 0 to 1 where 0 is topmost position and 1 is bottommost. For example, if a monitor is the target, a good choice is to set the topmost position to 0 and bottommost position to 1, i.e. the position unit will be percentage of a screen.
- nrofdata [IN]: Hint to the system of how many good samples to collect for this calibration point. A value of 6 is considered optimal for many eye tracking environments.

- Fnc [IN]: Pointer to a user defined callback function. Set NULL if callbacks aren't required.
- pApplicationData [IN]: Optional pointer to user defined data. This pointer will be passed as an argument to the callback function. Ignored if callback function is NULL.
- interval [IN]: Interval in milliseconds between calls to the callback function will be called. If set to 0, the callback will never be called for timer reason. Note that there's a lower limit of this interval dependent of the system the application runs on. The interval is not guaranteed to be followed strictly. See it more like a hint. Ignored if callback function is NULL

Modifying:
The calibration under construction.

```
long __stdcall Tet_CalibCalculateAndSet(void);
```

This call reads the calibration under construction and writes it to the calibration in use, which is immediately used by the eye tracking algorithm. Any old data is replaced. After this call there is an option to use Tet_CalibGetResult to inspect the samples to see if there are some improvements that can be made. It is not possible to inspect the samples before Tet_CalibCalculateAndSet is called. An error is returned if there are not enough samples in the calibration under construction. This may happen even though Tet_CalibAddPoint was called repeatedly. The cause may be that there is no subject to calibrate or if it is hard to track the subject.

Modifying:
The calibration in use.

```
long __stdcall Tet_CalibGetResult(char *pFile,
STet_CalibAnalyzeData *pData, long *pLen);
```

This function gets information about a calibration for a quality inspection. This inspection is easiest implemented as viewing the data to manually approve it or take necessary action to improve it. Provided are target points, the resulting mapped points and an indication whether the point was discarded or not, for some reason, by the eye tracker. Result may be retrieved from either the current calibration in use or from a calibration stored in a file (created by Tet_CalibSaveToFile). Memory to store the data to retrieve must be allocated by the caller. If 200 items are allocated, it will be enough for most calibrations.

Arguments:

- pFile [IN]: This parameter controls the calibration source to inspect. If set NULL, the source will be the current calibration in use. Otherwise, set it to a pointer to a null terminated array of chars that contains the path and filename to a calibration stored in a file by Tet_CalibSaveToFile.
- pData [IN]: Pointer to the first element in an array of STet_CalibAnalyzeData. The memory must be allocated by caller prior to call.
- pLen [IN/OUT]: In: number of items allocated in pData.
Out: number of items returned in pData. If data won't fit, pData is filled to its limit and pLen will point to the true length, making it possible to reallocate using the true length and make a new call.

```
long __stdcall Tet_CalibRemovePoints(ETet_Eye eye, float x,
float y, float radius);
```

The purpose of this function is to improve the calibration under construction by removing bad samples. It enables the deletion of samples for all calibration points within a specified area. The area is given by a circle; a point and a radius from that

point. The unit of the point is the same as was used for the (x, y) position when Tet_CalibAddPoint was called. To find out which points are bad, use Tet_CalibGetResult.

Arguments:

- eye [IN]: The eye(s) to remove calibration samples for x [IN]: Horizontal position of circle center. Same unit as was used when the points were added. See Tet_CalibAddPoint.

- y [IN]: Vertical position of circle center. Same unit as was used when the points were added. See Tet_CalibAddPoint.

- radius [IN]: The distance from point (x, y) defining the circle from within the calibration points will be removed. Same unit as x and y.

Modifying:

The calibration under construction.

```
typedef void (__stdcall *Tet_CallbackFunction)
(ETet_CallbackReason reason, void *pData, void
 *pApplicationData);
/* reason is GAZEDATA or TIMER (if interval is not 0), pData is a pointer to
STet_GazeData, and pApplicationData is the parameter passes into Tet_Start
function */
```

```
typedef struct _STet_GazeData {
    long timestamp_sec;
    long timestamp_microsec;
    float x_gazepos_lefteye; /* 0 .. 1, thus, undependable from */
    float y_gazepos_lefteye; /* screen resolution */
    float x_camerapos_lefteye;
    float y_camerapos_lefteye;
    float diameter_pupil_lefteye;
    float distance_lefteye;
    long validity_lefteye; /* 4 as valid, 0 as invalid */
    float x_gazepos_righteye;
    float y_gazepos_righteye;
    float x_camerapos_righteye;
    float y_camerapos_righteye;
    float diameter_pupil_righteye;
    float distance_righteye;
    long validity_righteye;
} STet_GazeData;
```

A table to reveal common functions of API

	Functions	LC EyeGaze	SR EyeLink	SMI iViewX	Tobii 1750
Main functions	Init/Connect	+	+	+	+
	Exit/Disconnect	+	+	+	+
	Start/Stop tracking		+	+	+
	GetData	+	+	+	

Calibration	All-in-one calibration function	+	+	+	+ (in high-level API)
	Custom calibration			+	+

Next table is to reveal common gaze data values and their types.

Data	LC EyeGaze	SR EyeLink (two eyes)	SMI iViewX	Tobii 1750 (two eyes)
Frame counter	+ (1/60 th , int)		+ (string)	
Validity/Found	+ (0/1, int)			+ (0..4, int)
Timestamp	+ (sec, double)	+ (ms, uint)	+ (ms, int64)	+ (sec/ms, int)
X/Y	+ (pixel, int)	+ (pixel, float)	+ (pixel, double)	+ (0..1, float)
Pupil size	+ (mm, float)	+ (?, float)	+ (X/Y/W/H) (?, double)	+ (mm, float)
Camera Timestamp	+ (sec, double)			
Camera X/Y	+ (mm, float)	+ (?, float)		+ (pixel, float)
Head reference X/Y		+ (?, float)		
Reflex X/Y			+ (?, double)	
Distance				+ (mm, float)

Note: the data structure of each API contains more members (sometimes much more, as EyeLink's API, for example). I put in this table only those that seems the most relevant for me.