ViewPoint-Presentation

a ***ViewPoint EyeTracker*** ® interface

to the ***NBS Presentation*** ® program

Software UserGuide

Arrington Research, Inc.

ViewPoint-Presentation UserGuide	2.9.5.111	31-October-2016
Tested with :		
ViewPoint-Presentation.dll	2.9.5.111	31-October-2016
VPX_InterApp_32.dll	2.9.5.111	30-October-2016
ViewPointClient_32.exe	2.9.5.111	31-October-2016
ViewPoint_64.exe	2.9.5.123	20-September-2016

Arrington Research, Inc.
27237 N 71st Place, Scottsdale, AZ 85262
United States of America
www.ArringtonResearch.com
Phone 480.985.5810
ViewPoint-EyeTracker@ArringtonResearch.com

Acknowledgements: Special thanks to Peter Pebler of Neurobehavioral Systems, Inc. for help with both the original and this version 2 interface.

Table of Contents

1. EyeTracker2 Interface

1.1. Important changes from previous versions

1.1.1. GazePoint

VPX_GetGazePointCorrected2 is used to get this data. This function returns the result of smoothing, binocular averaging, parallax correction, nudging, etc. For raw data, make sure that in the *ViewPoint* application the "Smoothing Points" is set to 1, all averaging, nudging, etc. is off.

1.1.2. Display Mode

Presentation automatically sets the Display Mode (screen width and height) internally to the EyeTracker2 interface when an eye_tracker object in the PCL program is created. Since EyeTracker1 returned X and Y position points in *ViewPoint* units, EyeTracker2 bypasses the functionality of Presentation and returns *ViewPoint* units.

If a user wants to use Presentation's Display Mode or set the Display Mode themselves, they can use a new function provided in the send_command PCL function. The new command has the format: "$SetDisplayMode w h", where w is the screen width and h is the screen height as an integer value. A sample calling sequence would be:

```
eyeTracker.send_command("$SetDisplayMode 640 480");
```

If w or h are set to 0, the Display Mode values set by Presentation during creation of an eye_tracker object are used. If w or h are set to 1, the *ViewPoint* units are used like in EyeTracker1. Any other values for w or h are used to scale the X and Y data points returned by *ViewPoint*.

1.1.3. PupilHeight

Both the pupilHeight and the pupilWidth are now normalized with respect to the EyeCamera window width, so they are now commensurable, e.g. pupilAspect can now be calculated directly as (pupilWidth/pupilHeight). Previously pupilHeight was normalized with respect to the window height.

1.1.4. PupilSize

The default pupil segmentation method is now rotated ellipse, so the terms width and height are no longer accurate. The width value is now always the major-axis (longer axis) of the elliptical fit to the pupil and the height is the minor-axis.

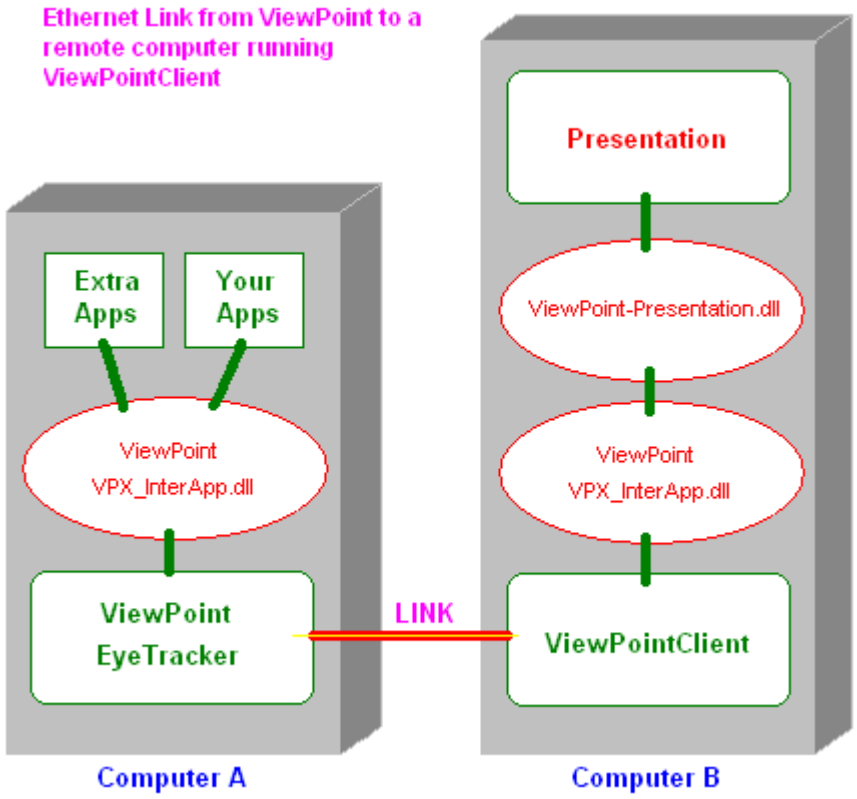
2. Introduction

The documentation describes a DLL based interface between ***ViewPoint EyeTracker ®*** (*ViewPoint*) and ***Presentation ®*** stimulus generation and experiment generation program.

Note that this documentation doesn't provide help for *Presentation* stimulus delivery and experimental control. The user should refer to the *Presentation* documentation for the same. It assumes that the user is familiar with the ***ViewPoint EyeTracker ®***.

2.1. ViewPoint-Presentation Interface

ViewPoint provides a generic interface to other processes (applications and programs) via a Dynamically Linked Library (DLL) based Software Developer Kit (SDK), named **VPX_InterApp.dll**. Because *Presentation* monopolizes the computer, *ViewPoint* must be run on a separate computer. *ViewPoint* also provides an easy way to communicate to the DLL based SDK running on another machine. A "light-weight" application called ***ViewPointClient ™*** runs on the second computer and communicates with *ViewPoint*. See figure below.



2.2. Presentation is a 32-bit application

Presentation is a 32-bit application, regardless of whether you downloaded the 32-bit or the 64-bit installation download. Consequently, you must use the VPX_InterApp_32.dll and the ViewPointClient_32.exe application.

3. Installing the interface.

3.1. Steps for installing the DLL interface

1. Put the file **ViewPoint-Presentation.dll** in the same folder as **VPX_InterApp.dll** and **ViewPoint.exe**.

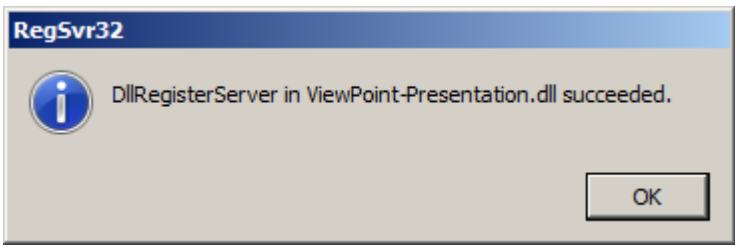
2. Open a command prompt, cmd.exe, as administrator in that folder and type:

regsvr32 ViewPoint-Presentation.dll

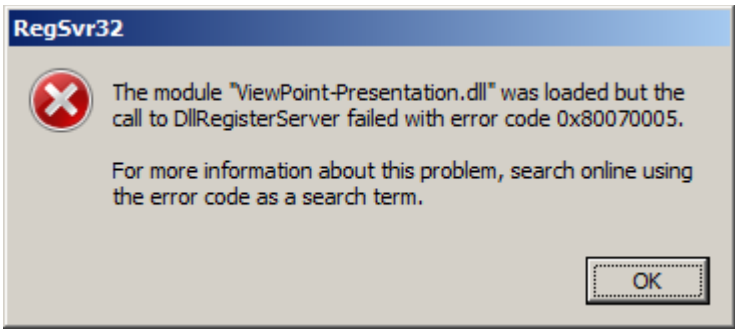
specifically:

a) **Start > Programs > Accessories > Command Prompt** (varies with OS).
b) **CD C:/ARI/ViewPoint** (or whatever your path is to the *ViewPoint* folder).
c) **regsvr32 ViewPoint-Presentation.dll**

You should see the following window:



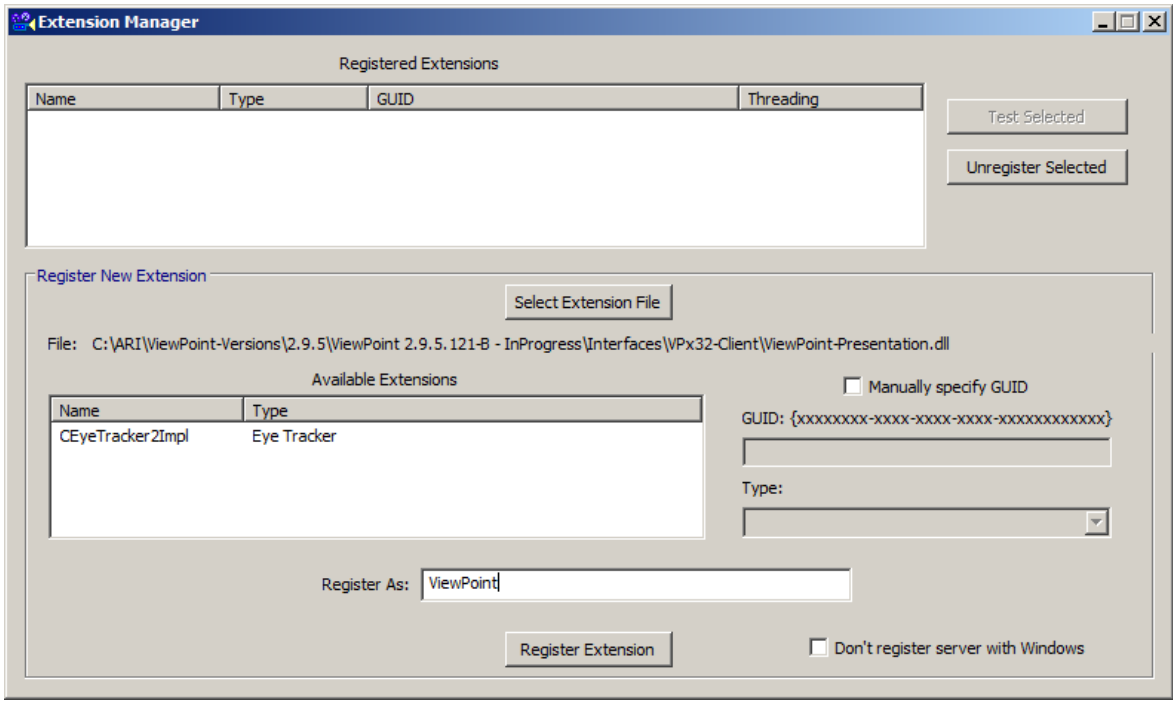
You must run **cmd.exe** as administrator, otherwise, you will probably see the following error:



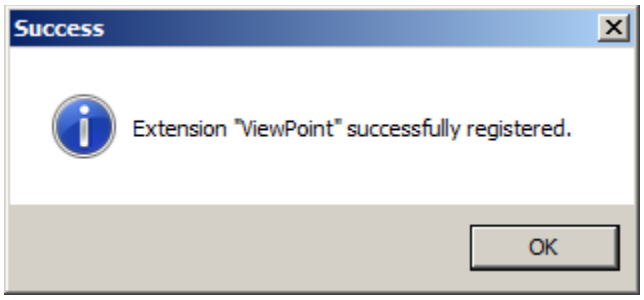
3. Run ***Presentation***.

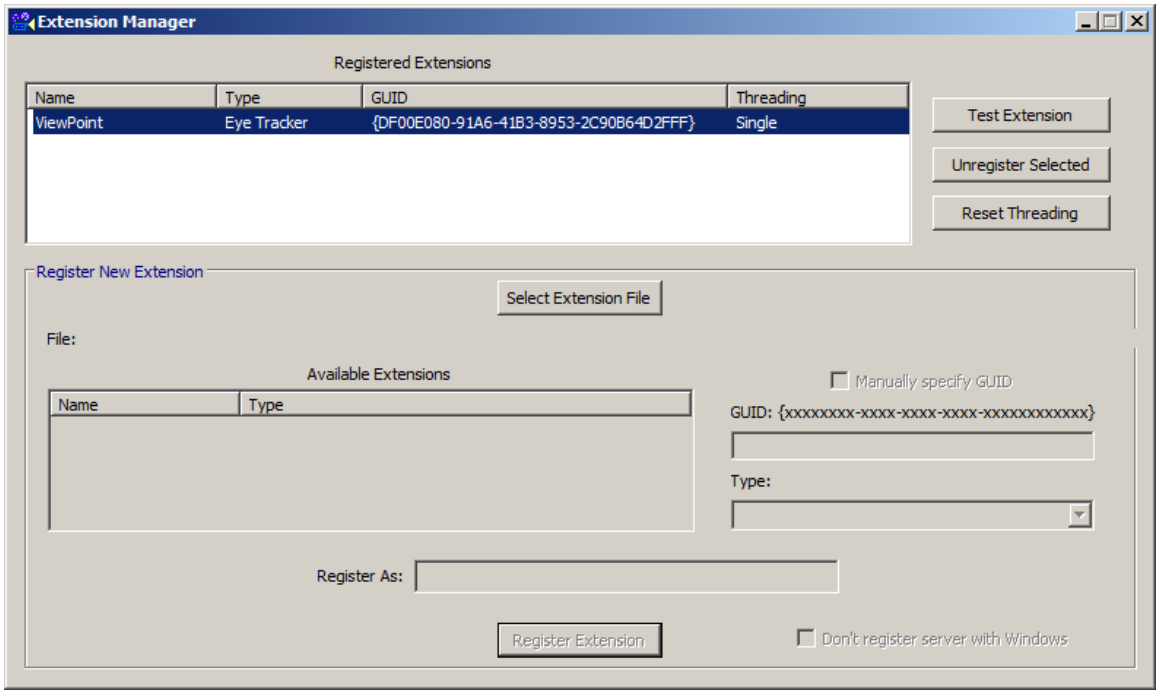
4. From the "**Tools**" menu, select "**Extension Manager**"

In the Register New Extension panel at the bottom, Press the [Select Extension File] button. This will bring up a window that you will use to navigate to and select the file: ViewPoint-Presentation.dll. After this is done, you will see a new entry under Available Extensions, which should show: CEyeTracker2Impl EyeTracker.



In the field: Register As: enter a name, such as ViewPoint, then press the [Register Extension] button; you should see a window indicating Success and the extension will appear in the top section in the Registered Extensions list view box.





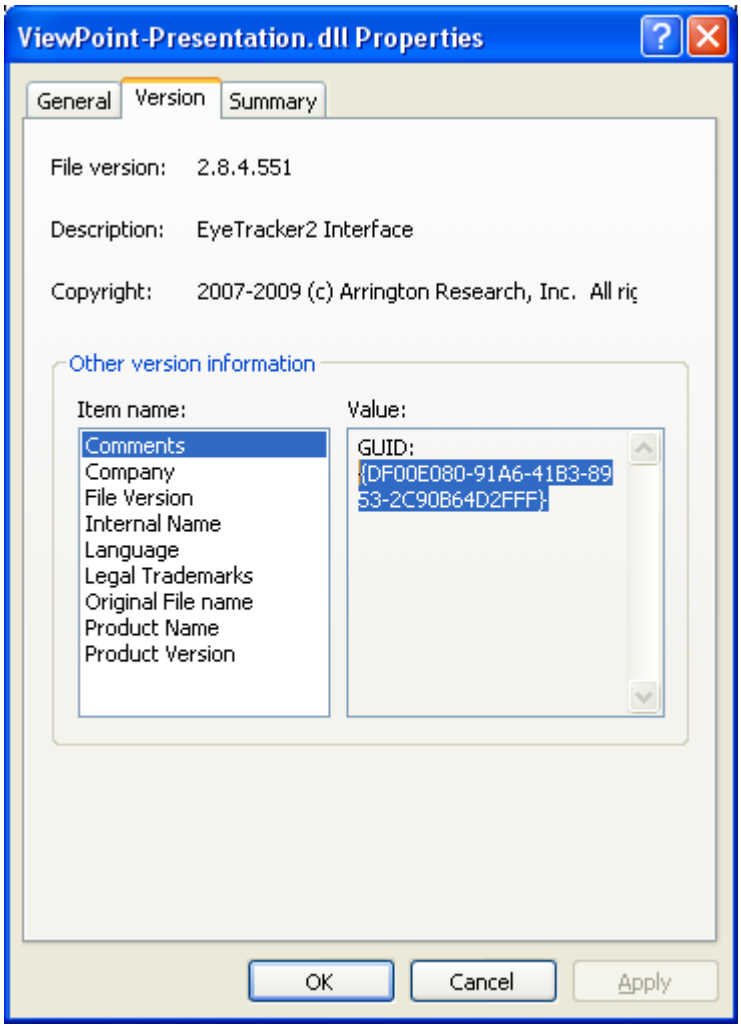
3.2. Testing the interface

Steps 1-5 in the previous section are performed for using the Eye Tracker interface with Presentation. Steps 6-8 in this section should be followed the first time to bring up a test dialog box that can be used for testing the Eye Tracker interface.

6. Select and test:
    a) Select that line newly created entry in the "Registered Extensions" list view.
    b) Click on the [**Test Selected**] button.

7. A new dialog window will appear.
    a) Click **Start** at the upper left corner, and then
    b) click **Start** in the "Tracking" panel.

8. Click "**Start**" in other panels to see that type of data.


ViewPoint-Presentation.dll Properties
General
Version
Summary
File version: 2.8.4.551
Description: EyeTracker2 Interface
Copyright: 2007-2009 (c) Arrington Research, Inc. All ric
Other version information
Item name:
Value:
Comments
Company
File Version
Internal Name
Language
Legal Trademarks
Original File name
Product Name
Product Version
GUID:
{DF00E080-91A6-41B3-8953-2C90B64D2FFF}
OK
Cancel
Apply

4. Getting Position-of-Gaze (POG) Data

Initially, the gaze position and pupil size will be in *ViewPoint* units (if the user is using the default pixel units in *Presentation*), except that **position is in the range -0.5 to 0.5, positive right and up**, such that:

```
Presentation_Position_X = ( VP_GazeSpace_X - 0.5 );
Presentation_Position_Y = ( 0.5 - VP_GazeSpace_Y );
```

To get the Presentation_Position_X and Presentation_Position_Y in pixels the user should multiply the Presentation_Position_X by (-screenwidth) and multiply Presentation_Position_Y by the screenheight. The following code snippet shows how to do this:

```
#-----------------------------------------------------------------------
eye_position_data position;
eye_tracker eyeTracker = new eye_tracker( "ViewPoint" );
#Start the tracking
eyeTracker.start_tracking();
eyeTracker.start_data(dt_position,true);
#Get the position data
position=eyeTracker.last_position_data();
#Obtain the screenwidth and screenheight
double ScreenWidth=double(display_device.width());
double ScreenHeight=double(display_device.height());
#Obtain the position in pixels
double position_x=position.x()*(-ScreenWidth);
double position_y=position.y()*ScreenHeight;
#-----------------------------------------------------------------------------------------------------------------------
```

5. Calibration

The ***eye_tracker::calibrate*** method is used to calibrate the *ViewPoint EyeTracker*. Four arguments have to be passed to the method for calibration. Please look at the method descriptions(under **EyeTracker methods)** for an explanation of the parameters that need to be passed for calibration.

There are hundreds of commands that can be sent to *ViewPoint* to control it via the SendCommand PCL function. The user may find it useful to send some of the following commands to *ViewPoint* for finer control over calibration:

calibration_Points *numberOfPoints*

Calibration_AutoIncrement *BoolValue*
***BoolValue*: Yes, No, True, False, On, Off, 1, 0, Toggle**

calibration_PresentationOrder *orderChoice*
***orderChoice*: Sequential, Random, Custom**

calibration_SnapMode *BoolValue*
***BoolValue*: Yes, No, True, False, On, Off, 1, 0, Toggle**

calibration_Snap

Snap&Inc

calibrationRedoPoint *N*

6. Interface methods

Optional eye arguments: Many `eye_tracker` methods have two versions: one with an `eye` argument and one without. This is indicated by the use of square brackets, `[]`, around the optional argument in the argument list for each method. If you include the `eye` argument, you will be requesting data for a particular eye. The value of this argument may be one of the pre-defined values `et_default`, `et_left`, `et_right`. In addition, you may use any other integer defined by your eye tracker extension. When using eye type `et_default`, your eye tracking software will determine what data to send. Version 1 eye tracker extensions only support eye type `et_default`. Therefore, there is no point to using the optional arguments with a version 1 eye tracker extension. Version 2 eye tracker extensions may support sending data for different eye designations simultaneously. (Use the `eye_tracker::version` method to find our which interface version your extension uses.) You may access the data for each eye using the optional `eye` arguments. You may also access one of the data sets using methods without the optional `eye` argument. Initially, doing so will access the data for eye type `et_default`. However, you can change this setting using the `eye_tracker::set_default_data_set` method.

Data Types: Some methods are used to adjust properties for data of all types. These methods take a `data_type` argument. Use one of the following pre-defined values for `data_type`: `dt_position`, `dt_saccade`, `dt_fixation`, `dt_aoi`, `dt_pupil`, `dt_blink`.

Error Handling: Some `eye_tracker` methods return a string that can be used to return an error message from the eye tracker extension. By default, any error returned by the extension will cause the scenario to abort. However, you may optionally choose to handle such errors in your PCL program by checking the return value for those methods. To do this, call the `eye_tracker::set_abort_on_error` method with an argument of `false`. In this case, an empty string returned from those methods indicates there was no error. If you choose to handle errors this way, make sure that you always check the return values. Note that other types of errors from the eye tracker extension may still cause the scenario to abort, and version 2 eye tracker extensions may still elect to abort the scenario for some types of errors even if the method has an error message return value.

6.1. Setup

6.1.1. new eye_tracker(string object_id) : eye_tracker

ViewPoint: Creates a new eye_tracker object and returns a reference to it.

Presentation: This method creates a new `eye_tracker` object and returns a reference to it. `object_id` can be either the GUID of the eye tracker extension (in the text format "{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}") or the friendly name given to the extension when it was registered with Presentation.

6.1.2. set_max_buffer_size([int eye,] int data_type, int size) : void

ViewPoint: Sets the maximum number of data samples of the specified data type that will be buffered in Presentation when eye_tracker::start_data(int, bool) is called with a store_data value of true or eye_tracker::start_data(int) is called. The maximum buffer size for all types is by default set to 100. The "size" argument must be between 1 and 100,000.The pre-defined values for data_type: dt_position, dt_saccade, dt_fixation, dt_aoi, dt_pupil, dt_blink.

Presentation: Sets the maximum number of data samples of the specified data type and eye that will be buffered in Presentation. The maximum buffer size for all types is by default set to 100. The `size` argument must be between 1 and 100,000. The buffers are circular, meaning that when a buffer is filled new data begins overwriting the oldest data. Calling this method clears the buffer of the specified type. See the notes above concerning the values of the `eye` and `data_type` arguments.

6.1.3. set_default_data_set(int eye) : void

ViewPoint: This method determines which data set is accessed when you use the data access methods without the optional `eye` argument. See the notes at the top concerning this method.

Presentation: This method determines which data set is accessed when you use the data access methods without the optional `eye` argument. See the notes at the

6.1.3. `set_default_data_set( int eye ) : void`

top concerning this method.

6.1.4. `set_abort_on_error( bool abort ) : void`

ViewPoint: See the notes on "Error Handling" at the top concerning this method.

Presentation: See the notes on "Error Handling" at the top concerning this method.

6.2. Data acquisition

6.2.1. start_data(int eye, int data_type, bool store_data) : string

ViewPoint: Instructs the eye tracker to start sending data of the specified type. If "store_data" is false, the eye_tracker object will only store the most recent data sample. Use one of the following predefined values for data_type: dt_position, dt_pupil.

Presentation: This method instructs the eye tracker to start sending data of the specified type for the specified eye. See the notes above concerning the values of the `data_type` and `eye` arguments. Note that the eye tracker will not automatically send eye data just because it is tracking. You must request the data you would like to receive. If the `store_data` argument is `false`, the `eye_tracker` object will only store the most recent data sample. If `store_data` is true, the `eye_tracker` object will store incoming data in a buffer. Presentation is not meant to be a data acquisition system; do not store large volumes of data as this data is stored in RAM. Any data of the specified type collected from a previous call to `eye_tracker::start_data` will be erased. See the notes on "Error Handling" at the top concerning the return value.

6.2.2. start_data(int data_type [, bool store_data]) : string

ViewPoint: Instructs the eye tracker to start sending data of the specified type. Use one of the following predefined values for data_type: dt_saccade, dt_fixation, dt_aoi, dt_blink.

Presentation: This method instructs the eye tracker to start sending data of the specified type. See the notes above concerning the value of the `data_type` argument. Note that the eye tracker will not automatically send eye data just because it is tracking. You must request the data you would like to receive. If the optional `store_data` argument is `false`, the `eye_tracker` object will only store the most recent data sample. If `store_data` is true, the `eye_tracker` object will store incoming data in a buffer. If you do not use this argument, its value will be `true`. Presentation is not meant to be a data acquisition system; do not store large volumes of data as this data is stored in RAM. Any data of the specified type collected from a previous call to `eye_tracker::start_data` will be erased. See the notes on "Error Handling" at the top concerning the return value. This method implicitly requests data for the eye type `et_default`. See the notes

6.2.2. `start_data( int data_type [, bool store_data] ) : string`

above.

6.2.3. `stop_data( [int eye,] int data_type ) : string`

ViewPoint: Instructs the eye tracker to stop sending data of the specified type. Use one of the following predefined values for data_type: dt_position, dt_saccade, dt_fixation, dt_aoi, dt_pupil, dt_blink.

Presentation: Instructs the eye tracker to stop sending data of the specified type and eye. See the notes above concerning the values of the `eye` and `data_type` arguments. See the notes on "Error Handling" at the top concerning the return value.

6.2.4. `clear_buffer( [int eye,] int data_type ) : void`

ViewPoint: Instructs Presentation to clear any stored data of the specified type. Use one of the following predefined values for data_type: dt_position, dt_saccade, dt_fixation, dt_aoi, dt_pupil, dt_blink.

Presentation: This method clears all data in the data buffer for the specified eye and data type. See the notes above concerning the values of the `data_type` and `eye` arguments.

6.2.5. event_count([int eye,] int data_type) : int

ViewPoint: Returns the number of events of the specified type that have been received since the buffer for the specified type was last cleared. The buffer for a given data type is cleared explicitly by a call to eye_tracker::clear_buffer or implicitly by calls to eye_tracker::start_data or eye_tracker::set_max_bufer_size. Use one of the following pre-defined values for data_type: dt_position, dt_saccade, dt_fixation, dt_aoi, dt_pupil, dt_blink.

Presentation: This method returns the number of events of the specified type and eye that have been received since the buffer for the specified type and eye was last cleared. See the notes above concerning the values of the `data_type` and `eye` arguments. The buffer for a given data type is cleared explicitly by a call to `eye_tracker::clear_buffer` or implicitly by calls to `eye_tracker::start_data` or `eye_tracker::set_max_buffer_size`. This value will increase as each new event is received even if the `eye_tracker::start_data` method is called with a `store_data` argument of `false`. In this case however, since Presentation is only storing the most recent data sample, the buffer will always contain only one item. This value will increase even if the buffer is full and wraps to the beginning.

6.3. Position Data

6.3.1. buffer_position([int eye,] int data_type) : int

ViewPoint: Returns the index of the last buffer slot to be filled with data of the specified type. If data of the specified types has been received, values will be between 1 and the maximum buffer size for the specified type. Use one of the following predefined values for data_type: dt_position, dt_saccade, dt_fixation, dt_aoi, dt_pupil, dt_blink.

Presentation: This method returns the index of the last buffer slot to be filled with data of the specified type and eye. See the notes above concerning the values of the `data_type` and `eye` arguments. If data of the specified types has been received, values will be between 1 and the maximum buffer size for the specified type. If no data has been received since the last call to `eye_tracker::start_data`, `eye_tracker::clear_buffer`, or `eye_tracker::set_max_buffer_size`, the return value will be 0. Buffer sizes for all types default to 100 unless explicitly set by a call to `eye_tracker::set_max_buffer_size`. The buffers are circular, so that when the buffer is filled, newer data will begin overwriting the data at the beginning of the buffer.

6.3.2. get_position_data([int eye,] int index) : eye_position_data

ViewPoint: Returns an eye_position_data with index "index" in the buffer of stored position data. If start_data was called with a "store_data" argument of false, the buffer will contain a maximum of 1 item. The index of the first item is 1.

Presentation: This method returns the `eye_position_data` object with index `index` in the position data buffer for the optionally specified eye. See the notes above concerning the optional `eye` argument. If the `eye_tracker::start_data` method was called with the `store_data` argument set to `false`, the buffer will contain a maximum of 1 item. The index of the first item in the buffer is 1.

6.3.3. last_position_data([int eye]) : eye_position_data

ViewPoint: Returns an eye_position_data reference containing the most recent eye position data received. It is an error to call this method if no data is available.

Presentation: This method returns an `eye_position_data` object for the most recent position data sample received for the specified eye. See the notes at the top concerning the optional `eye` argument. It is an error to call this method if no position data has been received since the last time the buffer was cleared.

6.3.4. new_position_data([int eye]) : int

ViewPoint: Returns the number of new position data has been received since the last call to this method. Calling start_data, clear_buffer or set_max_buffer_size for this data type resets this value to 0.

Presentation: This method returns the number of new position data samples that have been received since the last call to this method. See the notes at the top concerning the optional `eye` argument. Calling `eye_tracker::start_data`, `eye_tracker::clear_buffer` or `eye_tracker::set_max_buffer_size` for this data type resets this value to 0. The value returned by this method will increase as each new data sample is received, even if the `eye_tracker::start_data` method was called with the `store_data` argument set to false. If Presentation is not buffering multiple position data samples, the buffer will still only contain the most recent sample.

6.4. Trigger

6.4.1. get_trigger( ) : int

ViewPoint: Currently not used.

Presentation: Returns a numerical code received from the eye tracker. The return value of the `eye_tracker::trigger_count` method must be greater than 0 before calling this method. This method returns codes in the order in which they were received.

6.4.2. send_trigger(int code) : void

ViewPoint: Sends a numbered trigger to *ViewPoint*.

`Same as Send Command [ trigger n ].`

`Triggers are not currently used for anything by ViewPoint.`

Presentation: This method sends a trigger code to the eye tracker hardware. The effect is device dependent. The documentation of your eye tracker extension should indicate when to use which method, if at all.

6.4.3. trigger_count( ) : int

ViewPoint: Currently not used.

Presentation: This method returns the number of numerical codes received from the eye tracker that are ready to be retrieved. Use the `eye_tracker::get_trigger` method retrieve the codes. They must be read one by one.

6.5. Send

6.5.1. send_string(string message) : void

ViewPoint: Inserts a string in the data file.

`Same as Send Command [ dataFile_InsertString "string" ]`

Presentation: This method sends a string message to the eye tracker hardware. The effect is device dependent. The documentation of your eye tracker extension should indicate when to use which method, if at all.

6.5.2. send_command(string message) : int

ViewPoint: Sends a command message to *ViewPoint*. The command messages that are available for *ViewPoint* are documented in its **User Guide**.

`Same as Send Command [ "string" ]`

Presentation: This method sends a command message to the eye tracker hardware, and returns a response code from the eye tracker. The effect is device dependent. The documentation of your eye tracker extension should indicate when to use which method, if at all.

6.5.3. send_message(string message) : string

ViewPoint: CURRENTLY NOT IMPLEMENTED

Presentation: This method sends a string to the eye tracker extension and receives a response. Your eye tracker extension defines what messages are allowed, and the meaning of the response. This method differs from the `eye_tracker::send_string` method in that it receives a string response. The documentation of your eye tracker extension should indicate when to use which method, if at all. This method is only supported by version 2 eye tracker extensions.

6.6. Recording

6.6.1. set_recording(bool recording_on) : string

ViewPoint: Opens or resumes a data file when recording is on (i.e. bool=TRUE) and pauses a data file when recording is off (i.e bool=FALSE).

`If TRUE and no data file is open, it sends:` **`dataFile_NewUnique`**

`TRUE sends:` **`dataFile_Resume`**`, FALSE sends:` **`dataFile_Pause`**

`A better solution may be to send:` **`dataFile_Pause Toggle`**

`To close the dataFile send:` **`dataFile_Close`**

Presentation: This method instructs the eye tracker to record data (on the eye tracking computer) whenever it it tracking eye position. The effect is device dependent. See the notes on "Error Handling" at the top concerning the return value.

6.6.2. is_recording( ) : bool

ViewPoint: Indicates whether or not the eye tracker is storing data on the eye tracker computer.

Presentation: This method indicates whether or not the eye tracker is storing data on the eye tracker computer. This is independent of whether or not data is being sent to Presentation.

6.7. Tracking

6.7.1. start_tracking( ) : string

ViewPoint: Instructs the eye tracker to start tracking eye position data. This is independent of whether or not the data is being sent to Presentation.

Presentation: This method instructs the eye tracker to start tracking eye position data. This is independent of whether or not the data is being sent to Presentation. To receive data, you must also use the `eye_tracker::start_data` method. See the notes on "Error Handling" at the top concerning the return value.

6.7.2. stop_tracking( ) : string

ViewPoint: Instructs the eye tracker to stop tracking eye position data.

Presentation: This method instructs the eye tracker to stop tracking eye position data. See the notes on "Error Handling" at the top concerning the return value.

6.7.3. get_status( ) : int

ViewPoint: Retrieves a code that represents the status of the eye tracker hardware.

Presentation: This method retrieves a code that represents the status of the eye tracker hardware. The returned value will be equal to one of the following pre-defined PCL values: `et_status_stopped`, `et_status_initializing`, `et_status_tracking`, `et_status_error`. The meaning of these states is device-dependent. Check your vendor's documentation for details.

6.8. Supports

6.8.1. supports(int feature_code) : bool
ViewPoint: Returns a value specifying whether or not the eye tracker extension supports a given feature. The available features to query are provided as the following PCL pre-defined values: et_feature_position, et_feature_pupil_diameter, et_feature_fixation, et_feature_saccade, et_feature_aoi, et_feature_blink, et_feature_multiple_eyes.
Presentation: This method returns a value specifying whether or not the eye tracker extension supports a given feature. The available features to query are provided as the following PCL pre-defined values: `et_feature_position`, `et_feature_pupil`, `et_feature_fixation`, `et_feature_saccade`, `et_feature_aoi`, `et_feature_blink`, `et_feature_multiple_eyes`.

6.9. Calibration – Automatic

6.9.1. calibrate(int calibration_type, double parameter1, double parameter2, double parameter3) : string

ViewPoint: Instructs the eye tracker to perform a calibration operation.

calibration_type: 10

parameter1:

(p>0) : a positive value --> reset ViewPoint number of calibration points (NOTE: removes previous calibrations)

(p==-1) : a negative one --> do not respecify, just use the current ViewPoint specification.

parameter2:

(p>0) : a positive value --> calibrate only that point number.

(p==-1) : a negative one --> do sequential calibration

(p==-2) : a negative two --> do random calibration (currently locally randomized internal to the EyeTracker2 interface, not ViewPoint random sequence)

parameter3: TimeOut value in seconds.

e.g. 2.5: indicates a 2500 millisecond ISI for the calibration stimulus points.

CAUTION: parameter1, parameter2, and parameter3 must be specified as double, or an error will be produced by *Presentation*

Presentation: This method instructs the eye tracker to perform a calibration operation. `calibration_type` can be one of the predefined variables `et_calibrate_default` or `et_calibrate_drift_correct`, or an integer provided by the eye tracker manufacturer to designate a vendor-specific calibration routine. The additional parameters are eye tracker specific. See your eye tracker documentation for the use, if any, of these parameters. This method does not return until the calibration procedure has completed. If you define a picture stimulus named `et_calibration`, this picture stimulus will be used for the calibration display. The eye tracker extension controls the positioning of the picture parts in this stimulus during calibration. If you don't define a picture stimulus named `et_calibration`, Presentation will use a default white cross on a black background. The eye tracker extension can optionally use its own graphic for calibration, overriding the picture parts you put in your `et_calibration` picture. See the notes on "Error Handling" at the top concerning the return value.

6.10. Calibration – User Controlled

6.10.1. start_calibration(array<double,2> points, int eye, string parameters) : string

ViewPoint: CURRENTLY NOT IMPLEMENTED

Presentation: This method begins a calibration procedure during which your scenario will control the stimulus display. It is used in combination with the `eye_tracker::get_calibration_point`, `eye_tracker::accept_point`, and `eye_tracker::stop_calibration` methods. **If you want the eye tracker extension to manage the entire calibration procedure, including the stimulus display, use the `eye_tracker::calibrate` method instead**. Typically, you will only need to use this method if you require animations or sounds to be used for the calibration stimulus display. The `points` argument is a 2 dimensional array that can be used to request that specific calibration points be used. In this case, each element of the array is an array of size 2 containing the `x` and `y` coordinates of the points. For example, `points[1][1]` would be the `x` coordinate of the first point, and `points[1][2]` would be the `y` coordinate. The value should be in pixels, unless the scenario is using custom units. The origin is in the center of the screen with the `x` axis increasing to the right, and the `y` axis increasing upward. If the `points` array is empty, the eye tracker will determine which calibration points to use. See the notes at the top regarding the `eye` argument values. The `parameters` argument can be used to send additional arguments related to the calibration procedure. What arguments are allowed and what the format of this string should be are determined by your eye tracker extension. See the notes on "Error Handling" at the top concerning the return value. This method is only supported by version 2 eye tracker extensions.

6.10.2. stop_calibration( ) : string

ViewPoint: CURRENTLY NOT IMPLEMENTED

Presentation: This method can be used to instruct the eye tracker to stop the calibration procedure started using the `eye_tracker::start_calibration` method. Note that you should also call this method after receiving 0 for the calibration point index in a call to `eye_tracker::get_calibration_point`. See the notes on "Error Handling" at the top concerning the return value.

6.10.3. accept_point( ) : string

ViewPoint: CURRENTLY NOT IMPLEMENTED

Presentation: This method can be used during a manual calibration procedure to instruct the eye tracker to proceed to the next calibration point. You must call the `eye_tracker::start_calibration` method before using this method. Note that if the eye tracker or user interaction at the eye tracking computer is managing the progression of the calibration procedure, this method is not used. Use the `eye_tracker::get_calibration_point` method to get the location of the current calibration point. See the notes on "Error Handling" at the top concerning the return value.

6.10.4. get_calibration_point(int& index, double& x, double& y) : string

ViewPoint: CURRENTLY NOT IMPLEMENTED

Presentation: This method is used during a manual calibration procedure to obtain the location of the current calibration point. You must call the `eye_tracker::start_calibration` method before using this method. You should assume that the calibration procedure can move to the next point at any time, not just because `eye_tracker::accept_point` is called, so you should poll this method continuously during the calibration display. The `index` argument is set to the index of the current calibration point, where the first calibration point has index 1. If the calibration has ended, 0 is returned in this argument. Note that you should still call `eye_tracker::stop_calibration` even after receiving 0 for the calibration point index. The `x` and `y` arguments are set to the coordinates of the calibration point. These values will be in pixels, unless the scenario is using custom units. See the notes on "Error Handling" at the top concerning the return value.

6.11. Saccade events

6.11.1. new_saccade_events([int eye]) : int

ViewPoint: Returns the number of new saccade events that have been received since the last call to this method. Calling start_data, clear_buffer or set_max_buffer_size for this data type resets this value to 0.

Presentation: This method returns the number of new saccade data samples that have been received since the last call to this method. See the notes at the top concerning the optional `eye` argument. Calling `eye_tracker::start_data`, `eye_tracker::clear_buffer` or `eye_tracker::set_max_buffer_size` for this data type resets this value to 0. The value returned by this method will increase as each new data sample is received, even if the `eye_tracker::start_data` method was called with the `store_data` argument set to false. If Presentation is not buffering multiple data samples, the buffer will still only contain the most recent sample.

6.11.2. get_saccade_event([int eye,] int index) : saccade_event_data

ViewPoint: Returns saccade_event_data with index "index" in the buffer of stored saccade data. Call event_count to find out how many items have been received since the last call to start_data, clear_buffer or set_max_buffer_size for this data type. The index of the first item is 1.

Presentation: This method retrieves a code that represents the status of the eye tracker hardware. The returned value will be equal to one of the following pre-defined PCL values: `et_status_stopped`, `et_status_initializing`, `et_status_tracking`, `et_status_error`. The meaning of these states is device-dependent. Check your vendor's documentation for details.

6.11.3. last_saccade_event([int eye]) : saccade_event_data

ViewPoint: Returns a saccade_event_data reference containing the most recent saccade event received. It is an error to call this method if no data is available.

Presentation: This method returns a `saccade_event_data` object for the most recent saccade event received for the specified eye. See the notes at the top concerning the optional `eye` argument. It is an error to call this method if no events have been received since the last time the buffer was cleared.

6.12. Fixation

6.12.1. new_fixation_events([int eye]) : int

ViewPoint: Returns the number of new fixation events that have been received since the last call to this method. Calling start_data, clear_buffer or set_max_buffer_size for this data type resets this value to 0.

Presentation: This method returns the number of new fixation data samples that have been received since the last call to this method. See the notes at the top concerning the optional `eye` argument. Calling `eye_tracker::start_data`, `eye_tracker::clear_buffer` or `eye_tracker::set_max_buffer_size` for this data type resets this value to 0. The value returned by this method will increase as each new data sample is received, even if the `eye_tracker::start_data` method was called with the `store_data` argument set to false. If Presentation is not buffering multiple data samples, the buffer will still only contain the most recent sample.

6.12.2. get_fixation_event([int eye,] int index) : fixation_event_data

ViewPoint: Returns fixation_event_data with index "index" in the buffer of stored fixation event data. Call event_count to find out how many items have been received since the last call to start_data, clear_buffer or set_max_buffer_size for this data type. The index of the first item is 1

Presentation: This method returns the `fixation_event_data` object with index `index` in the fixation data buffer for the optionally specified eye. See the notes above concerning the optional `eye` argument. If the `eye_tracker::start_data` method was called with the `store_data` argument set to `false`, the buffer will contain a maximum of 1 item. The index of the first item in the buffer is 1.

6.12.3. last_fixation_event([int eye]) : fixation_event_data

ViewPoint: Returns a fixation_event_data reference containing the most recent fixation event data received. It is an error to call this method if no data is available.

Presentation: This method returns a `fixation_event_data` object for the most recent fixation event received for the specified eye. See the notes at the top concerning the optional `eye` argument. It is an error to call this method if no events have been received since the last time the buffer was cleared.

6.13. Area Of Interest (AOI)

ViewPoint uses negative AOI numbers to indicate when an AOI is exited; do not use deprecated AOI#0 (zero).

Hint: For better viewing of AOI (ROI) changes within Presentation's *Eye Tracker Extension Test* dialog box, set *ViewPoint*'s ROI #99 to the full gazeSpace window. This can be used to detect out of AOI and re-entry.

6.13.1. new_aoi_events([int eye]) : int

ViewPoint: Returns the number of new aoi events that have been received since the last call to this method. Calling start_data, clear_buffer or set_max_buffer_size for this data type resets this value to 0.

Presentation: This method returns the number of new area of interest event data samples that have been received since the last call to this method. See the notes at the top concerning the optional `eye` argument. Calling `eye_tracker::start_data`, `eye_tracker::clear_buffer` or `eye_tracker::set_max_buffer_size` for this data type resets this value to 0. The value returned by this method will increase as each new data sample is received, even if the `eye_tracker::start_data` method was called with the `store_data` argument set to false. If Presentation is not buffering multiple data samples, the buffer will still only contain the most recent sample.

6.13.2. get_aoi_event([int eye,] int index) : aoi_event_data

ViewPoint: Returns aoi_event_data with index "index" in the buffer of stored aoi event data. Call event_count to find out how many items have been received since the last call to start_data, clear_buffer or set_max_buffer_size for this data type. The index of the first item is 1.

Presentation: This method returns the `aoi_event_data` object with index `index` in the area of interest data buffer for the optionally specified eye. See the notes above concerning the optional `eye` argument. If the `eye_tracker::start_data` method was called with the `store_data` argument set to `false`, the buffer will contain a maximum of 1 item. The index of the first item in the buffer is 1.

6.13.3. last_aoi_event([int eye]) : aoi_event_data

ViewPoint: Returns an aoi_event_data reference containing the most recent aoi event data received. It is an error to call this method if no data is available.

Presentation: This method returns a `aoi_event_data` object for the most recent area of interest event received for the specified eye. See the notes at the top concerning the optional `eye` argument. It is an error to call this method if no events have been received since the last time the buffer was cleared.

6.13.4. set_aoi_set(int aoi_set) : string

ViewPoint: `Loads the set of AOI specifications saved in ~ViewPoint/Settings/ROI/n.txt, where n is the specified group number.`

`Same as Send Command [ setROI_group n ].`

Presentation:: This method instructs the eye tracking software to use a particular area of interest map to generate area of interest event data. The argument values and use of this method are device dependent. See the notes on "Error Handling" at the top concerning this method.

6.14. Pupil

6.14.1. new_pupil_data([int eye]) : int

ViewPoint: Returns the number of new pupil data have been received since the last call to this method. Calling start_data, clear_buffer or set_max_buffer_size for this data type resets this value to 0.

Presentation: : This method returns the number of new pupil data samples that have been received since the last call to this method. See the notes at the top concerning the optional `eye` argument. Calling `eye_tracker::start_data`, `eye_tracker::clear_buffer` or `eye_tracker::set_max_buffer_size` for this data type resets this value to 0. The value returned by this method will increase as each new data sample is received, even if the `eye_tracker::start_data` method was called with the `store_data` argument set to false. If Presentation is not buffering multiple data samples, the buffer will still only contain the most recent sample.

6.14.2. get_pupil_data([int eye,] int index) : pupil_data

ViewPoint: Returns pupil_data with index "index" in the buffer of stored pupil data. Call event_count to find out how many items have been received since the last call to start_data, clear_buffer or set_max_buffer_size for this data type. If start_data was called with a "store_data" argument of false, the buffer will contain a maximum of 1 item. The index of the first item is 1.

Presentation: This method returns the `pupil_data` object with index `index` in the pupil data buffer for the optionally specified eye. See the notes above concerning the optional `eye` argument. If the `eye_tracker::start_data` method was called with the `store_data` argument set to `false`, the buffer will contain a maximum of 1 item. The index of the first item in the buffer is 1.

6.14.3. last_pupil_data([int eye]) : pupil_data

ViewPoint: Returns a pupil_data reference containing the most recent eye position data received. It is an error to call this method if no data is available.

Presentation: This method returns an `pupil_data` object for the most recent pupil data sample received for the specified eye. See the notes at the top

6.14.3. last_pupil_data([int eye]) : pupil_data

concerning the optional `eye` argument. It is an error to call this method if no pupil data has been received since the last time the buffer was cleared.

6.15. Blink

6.15.1. new_blink_events([int eye]) : int

ViewPoint: Returns the number of new blink events that have been received since the last call to this method.

Presentation: This method returns the number of new blink event data samples that have been received since the last call to this method. See the notes at the top concerning the optional `eye` argument. Calling `eye_tracker::start_data`, `eye_tracker::clear_buffer` or `eye_tracker::set_max_buffer_size` for this data type resets this value to 0. The value returned by this method will increase as each new data sample is received, even if the `eye_tracker::start_data` method was called with the `store_data` argument set to false. If Presentation is not buffering multiple data samples, the buffer will still only contain the most recent sample.

6.15.2. get_blink_event([int eye,] int index) : blink_event_data

ViewPoint: Returns blink_event_data with index "index" in the buffer of stored blink event data. Call event_count to find out how many items have been received since the last call to start_data, clear_buffer or set_max_buffer_size for this data type. The index of the first item is 1.

Presentation: This method returns the `blink_event_data` object with index `index` in the blink event buffer for the optionally specified eye. See the notes above concerning the optional `eye` argument. If the `eye_tracker::start_data` method was called with the `store_data` argument set to `false`, the buffer will contain a maximum of 1 item. The index of the first item in the buffer is 1.

6.15.3. last_blink_event([int eye]) : blink_event_data

ViewPoint Specific Error message: last_blink_event not implemented

Presentation: This method returns a `blink_event_data` object for the most recent blink event received for the specified eye. See the notes at the top concerning the optional `eye` argument. It is an error to call this method if no events have been received since the last time the buffer was cleared.

6.16. Parameters

6.16.1. get_parameter(string name) : string

ViewPoint: CURRENTLY NOT IMPLEMENTED

Presentation: This method returns the value of the named parameter. Acceptable parameter names are defined by your eye tracker extension. If the returned value is a number, use the `int` or `double` functions to convert the value. This method is only supported by version 2 eye tracker extensions.

6.16.2. set_parameter(string name, string value) : string

ViewPoint: CURRENTLY NOT IMPLEMENTED

Presentation: This method sets an eye tracker parameter to the specified value. Your eye tracker extension defines what parameters may be set. If the parameter value is a number in a variable, use the `string` function to convert it to a string. See the notes on "Error Handling" at the top concerning the return value. This method is only supported by version 2 eye tracker extensions.

6.17. Version

6.17.1. version( ) : int
ViewPoint: This method returns the interface version of the eye tracker extension.
Presentation: This method returns the interface version of the eye tracker extension.

7. Methods for accessing elements of returned data.

Methods in **eye_position_data**
time() : int Returns the timestamp of the associated position data in milliseconds converted to Presentation time.
x() : double Returns the x coordinate of the eye position in pixels, unless you are using the `screen_distance` or `max_y` header parameters, in which case your own custom units are used. The origin is the center of the screen and positive is to the right.
y() : double Returns the y coordinate of the eye position in pixels, unless you are using the `screen_distance` or `max_y` header parameters, in which case your own custom units are used. The origin is the center of the screen and positive is upward.

Methods in **saccade_event_data**
time() : int Returns the timestamp of the associated saccade event in milliseconds converted to Presentation time.
x() : double Returns the x coordinate of the saccade event in pixels, unless you are using the `screen_distance` or `max_y` header parameters, in which case your own custom units are used. The origin is the center of the screen and positive is to the right.
y() : double Returns the y coordinate of the saccade event in pixels, unless you are using the `screen_distance` or `max_y` header parameters, in which case your own custom units are used. The origin is the center of the screen and positive is upward.
type() : int Returns the saccade event type saccade event. The data will be one of the predefined PCL values et_saccade_start or et_saccade_stop

Methods in **fixation_event_data**
time() : int Returns the timestamp of the associated fixation event in milliseconds converted to Presentation time.
x() : double Returns the x coordinate of the eye position in pixels, unless you are using the `screen_distance` or `max_y` header parameters, in which case your own custom units are used.. The origin is the center of the screen and positive is to the right.
y() : double Returns the y coordinate of the eye position in pixels, unless you are using the `screen_distance` or `max_y` header parameters, in which case your own custom units are used.. The origin is the center of the screen and positive is upward.
type() : int Returns the fixation event type. The value will be one of the predefined PCL values et_fixation_start or et_fixation_stop.

Methods in **aoi_event_data**
time() : int Returns the timestamp of the associated AOI event in milliseconds converted to Presentation time.
area() : int Returns the an AOI index associated with the AOI event. The return value will be one of the area-of-interest areas setup using the eye tracker's software. See your vendor's documentation for instructions on setting up areas of interest. ***Note:*** When eye moves into a defined AOI, the area is positive indicating the new AOI the gazePoint has entered. A negative area indicates that the gazePoint has just exited that AOI.

Methods in **pupil_data**
time() : int Returns the timestamp of the associated pupil data in milliseconds converted to Presentation time.
x_diameter() : double Returns the x diameter of the subject's pupil in pixels, unless you are using the `screen_distance` or `max_y` header parameters, in which case your own custom units are used.
y_diameter() : double Returns the y coordinate of the subject's pupil in pixels, unless you are using the `screen_distance` or `max_y` header parameters, in which case your own custom units are used.

8. Sample Code Examples

Some of the samples require making buttons active so please read all of the documentation at the top of each sample before running them. Also, the samples create an eye_tracker object using the code: new eye_tracker("ViewPoint");. The “ViewPoint” string must match the name that was entered into the “Register as” text box when the ViewPoint-Presentation.dll was registered with the Presentation Extension Manager.

8.1. Sample code 1

```
#-----------------------------------------------------------------------------------------------------
# AOI Events Sample Program 1
#-----------------------------------------------------------------------------------------------------

#-----------------------------------------------------------------------------------------------------
# The picture stimulus changes color as the eye moves.  When the eye is in the AOI, the picture stimulus
# is "blue" and when the eye is not in the AOI, the picture stimulus is "green".
#-----------------------------------------------------------------------------------------------------

# You must make 1 button active.
# To make the "Enter" button active, go to the "Settings" tab and select the "Response" button on the left.
# In the "Scenarios" box select this scenario.  In the "Active Buttons" list view, make sure there are no
# buttons active by pressing the "Clear" button to the right of the list.  In the "Devices" box, select the
# "Keyboard" item.  In the "Buttons" box, scroll down and select the "Enter" item.  Press the "Use" button
# to the right to add the "Enter" button to the "Active Buttons" list.
active_buttons = 1;
button_codes = 1;

begin;

# Initial caption.
trial
{
	trial_type = first_response;
	picture
	{
		text
		{
			system_memory = true;
			caption = "The movement of the eye moves the box in this scenario.\n\nPlease Note: This
scenario will last for 60 seconds.\n\nPress Esc to Stop.\nPress Enter to Start...";
			font_size = 24;
		};
		x = 0;
		y = 0;
	};
	time = 0;
	duration = response;
} my_trial;

# Displayed when eye is not in AOI (Green).
picture
{
	box
	{
		height = 75;
		width = 75;
		color = 0,255,0;
	};
	x = 0;
	y = 0;
} pic_notAOI;
```

```
# Displayed when eye is in AOI (Blue).
picture
{
	box
	{
		height = 75;
		width = 75;
		color = 0,0,255;
	};
	x = 0;
	y = 0;
} pic_AOI;

# Beginning of the PCL program.
begin_pcl;

# Present the initial caption.
my_trial.present();

# Get the screen width and height.
int w = display_device.width();
int h = display_device.height();
double screenWidth = double(w);
double screenHeight = double(h);

# Create a new EyeTracker object.  The object_id parameter should be the same "name"
# you registered your EyeTracker Extension with in the Extension Manager.
eye_tracker eyeTracker = new eye_tracker("ViewPoint");

# Start tracking.
eyeTracker.start_tracking();

# Start storing the Position and AOI Data.
eyeTracker.start_data(dt_position, true);
eyeTracker.start_data(dt_aoi);

# Loop for 1 minute (60 seconds) while showing the AOI Events.
loop
	int endTime = clock.time() + 60000; # Units are milliseconds
until
	clock.time() > endTime
begin
	# Check for new AOI Events.
	int aoiEventCount = eyeTracker.new_aoi_events();
	if(aoiEventCount > 0) then
		# Check for new Position Data.
		int posDataCount = eyeTracker.new_position_data();
		if(posDataCount > 0) then
			# Get the Position Data and scale to Screen coordinates.
			eye_position_data position = eyeTracker.last_position_data();
			double x = position.x() * (-screenWidth);
			double y = position.y() * screenHeight;

			# Get the last AOI Event.
			aoi_event_data aoiEvent = eyeTracker.last_aoi_event();

			# Get the area of the AOI.  If > 0 then AOI, else NOT AOI.
			int InAOI = aoiEvent.area();
			if(InAOI > 0) then
				# Display the AIO box.
				pic_AOI.set_part_x(1, x);
				pic_AOI.set_part_y(1, y);
				pic_AOI.present();
			else
				# Display the non AIO box.
				pic_notAOI.set_part_x(1, x);
				pic_notAOI.set_part_y(1, y);
				pic_notAOI.present();
```

```
                        end;
                    end;
                end;
            end;

#End of the sample code
#-----------------------------------------------------------------------------------------------------------------------
#-----------------------------------------------------------------------------------------------------------------------
```

8.2. Sample Code 2

```
#-----------------------------------------------------------------------------------------------
# AOI Events Sample Program 2
#-----------------------------------------------------------------------------------------------

#-----------------------------------------------------------------------------------------------
# This program captures AOI Events for 10 seconds are writes them to a text file.
#-----------------------------------------------------------------------------------------------

# You must make 1 button active.
# To make the "Enter" button active, go to the "Settings" tab and select the "Response" button on the left.
# In the "Scenarios" box select this scenario.  In the "Active Buttons" list view, make sure there are no
# buttons active by pressing the "Clear" button to the right of the list.  In the "Devices" box, select the
# "Keyboard" item.  In the "Buttons" box, scroll down and select the "Enter" item.  Press the "Use" button
# to the right to add the "Enter" button to the "Active Buttons" list.
active_buttons = 1;
button_codes = 1;

begin;

# Initial caption.
trial
{
	trial_type = first_response;
	picture
	{
		text
		{
			system_memory = true;
			caption = "This scenario will capture AOI events for 10 seconds and write them to a text
file \"aoi_area.txt\".\n\nPlease Note:\n\nPress Esc to Stop.\nPress Enter to Start...";
			font_size = 24;
		};
		x = 0;
		y = 0;
	};
	time = 0;
	duration = response;
} my_trial;

# Capturing caption.
picture
{
	text
	{
		system_memory = true;
		caption = "Capturing AOI Events...";
		font_size = 24;
	};
	x = 0;
	y = 0;
} pic_Capturing;

# Beginning of the PCL program.
begin_pcl;

# Present the initial caption.
my_trial.present();

# Present the capturing status message.
pic_Capturing.present();

# Create a new EyeTracker object.  The object_id parameter should be the same "name"
# you registered your EyeTracker Extension with in the Extension Manager.
eye_tracker eyeTracker = new eye_tracker("ViewPoint");
```

```
# Start tracking.
eyeTracker.start_tracking();

# Start storing the AOI Data.
eyeTracker.start_data(dt_aoi);

# Create the arrays that will hold the AOI Event Data.
array<int> aoiArea[0];
array<int> aoiTime[0];

# Loop for 10 seconds while capturing the AOI Event Data.
loop
	int endTime = clock.time() + 10000; # Units are milliseconds
until
	clock.time() > endTime
begin
	# Check for new AOI Events.
	int aoiEventCount = eyeTracker.new_aoi_events();
	if(aoiEventCount > 0) then
		# Get the last AOI Event.
		aoi_event_data aoiEvent = eyeTracker.last_aoi_event();

		# Set the area and time of the AOI Event.
		aoiArea.add(aoiEvent.area());
		aoiTime.add(aoiEvent.time());
	end;
end;

# Create and open a text file.  The file will be located in Presentation's "example" folder.
output_file out = new output_file;
out.open("aoi_area.txt");

# Loop through all the AOI Events and write them to the file.
loop
	int index = 1; # Arrays are 1-based.
	int count = aoiArea.count();
until
	index > count
begin
	# Write the Area and Time.
  out.print(aoiArea[index]);
  out.print("\t");
  out.print(aoiTime[index]);
  out.print("\n");

	# Make sure we advance to the next AOI Event.
	index = index + 1;
end;

# Make sure to close the text file.
out.close();

#End of the sample code
#-----------------------------------------------------------------------------------------------------------------------
#-----------------------------------------------------------------------------------------------------------------------
```

8.3. Sample Code 3

```
#-----------------------------------------------------------------------------------------------

# Calibration Sample Program 3
#-----------------------------------------------------------------------------------------------

#-----------------------------------------------------------------------------------------------
# This program calibrates 12 points sequentially displaying a white cross for each calibration point.
#-----------------------------------------------------------------------------------------------

begin;

# The default picture stimulus of the presentation is a white cross.
# If you want to have your own picture stimulus then it has to be created
# with name "et_calibration" like shown below.
picture
{
        box
        {
                height = 75;
                width = 75;
                color = 0,255,0;
        };

        x = 0;
        y = 0;
} pic;

# Beginning of the PCL program.
begin_pcl;

# Create a new EyeTracker object.  The object_id parameter should be the same "name"
# you registered your EyeTracker Extension with in the Extension Manager.
eye_tracker eyeTracker = new eye_tracker("ViewPoint2");

#Call the calibrate function
#calibrate function parameters
#1. Calibration type it has be set to 10
#2. parameter1 : number of stimulus points
#   (b>0) : a positive value --> reset ViewPoint number of calibration points (NOTE: removes previous
calibrations)
#   (b==-1) : a negative one --> do not respecify, just use the current ViewPoint specification.
#3. parameter2 : auto calibrate(-1) or point calibrate(point number to be calibrated)
#   (b>0) : a positive value --> calibrate only that point number.
#   (b==-1) : a negative one --> do sequential calibration
#   (b==-2) : a negative two --> do random calibration (currently locally randomized internal to the
EyeTracker2 interface, not ViewPoint random sequence)
#4. parameter3 : timeOut value in seconds

# Calibrates 12 points sequentially.
eyeTracker.calibrate(10,12.0,-1.0,2.0);

# Calibrates 12 points randomly.
#eyeTracker.calibrate(10,12.0,-2.0,2.0);

# Calibrates only point 1 (single point).  Does not reset the calibration data
#eyeTracker.calibrate(10,-1.0,1.0,2.0);
```