ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2008, Vol. 37, No. 1

ICOMPONENT: SOFTWARE WITH FLEXIBLE ARCHITECTURE FOR DEVELOPING PLUG-IN MODULES FOR EYE TRACKERS

Oleg Špakov

Department of Computer Sciences, University of Tampere
FIN-33014 Tampere, Finland

Darius Miniotas

Department of Electronic Systems, Vilnius Gediminas Technical University
LT-03227 Vilnius, Lithuania

Abstract. iComponent is a software tool we have developed to facilitate basic research on eye movements as well as applications of eye gaze for computer input. Despite the variety of eye trackers and gaze-data analysis tools available today, there is still a gap between what researchers need and what products are available on the market to suit their needs. Also, researchers are confronted with many difficulties due to complexity of the technology of eye tracking devices. We approached the need to standardize the input process by proposing a generic data format and making the software architecture flexible. This is achieved by inserting an intermediate software layer (API converter) between the core of iComponent and the API of the device attached.

Keywords: eye tracking, gaze-data analysis, gaze-path visualization, input device API converter, standardization.

1. Introduction

The growing number of publications that address human-computer interaction using eye movements highlights the increasing need for tools to investigate and analyze the behavior of human eyes [2, 3, 4, 6]. The first such tools were developed at roughly the same time as the first eye trackers became available, and they were designed according to the structure of the data produced by the eye trackers, with each tool supporting only one particular eye tracker. This fact hampered progress in moving toward use of newer devices, and a great deal of effort is required to port the functionality of the tools developed to newer platforms.

Today, many commercial and academic products are available for researchers in this field and the quality and accuracy of eye tracking devices are constantly increasing. Several researchers have made attempts to develop tools supporting analysis of data gathered by different eye trackers. At the same time, eye movement analysis tools are becoming more intelligent and advanced, and some manufacturers of eye tracking devices made a great shift to satisfy the requirements of the developers of the tools.

Nevertheless, there is still a lack of effective tools to allow recording and on-line usage of eye movement data from many eye trackers. Usable tools to support analysis and visualization of gaze paths by using the majority of methods already known are also scarce [7].

To address this problem, we developed the iComponent software having a highly flexible architecture to allow easy development of dynamic plug-in modules for eye tracking devices and experimental software.

2. Architecture

The iComponent software runs under the Windows 2000/XP operating system. It consists of the core (main window, dialogs, and panels), plug-in device programming interface converters, experiment plug-in modules, and a library of visualizations. iComponent may be in one of three states: recording eye movement (probably running some 'experiment' – see below), visualizing gaze path from a single recording, or comparing recordings made during the same study (i.e., using the same experiment module with the same set-up).

Each device programming interface converter corresponds to a particular eye tracker. The converters utilize device APIs provided by manufacturers to receive data and eye movement events as well as to control the eye trackers. All converters have the same functions (interface) available from the iComponent

core. In other words, the converters are dynamic libraries with different input interfaces but the same output interface. One module of the core uses the same converter attached to it during the experimental set-up and then during the data recording session that follows. Figure 1 shows a diagram of the connections between eye trackers and iComponent.

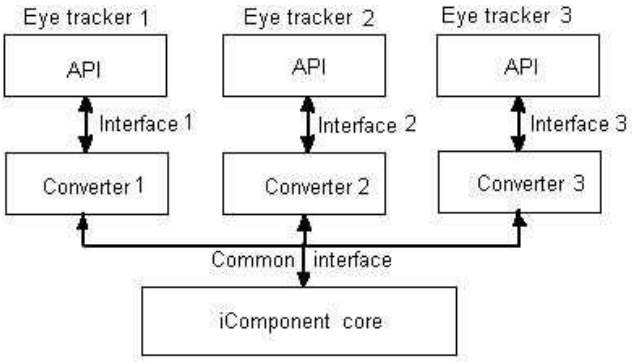

Figure 1. Device API interface converters

A device plug-in is a module providing a common interface for managing eye tracker and recording data to a file – here, referred to as an 'eye tracker driver API converter', or simply an API converter. There is one additional converter to allow simulation of gaze data using a conventional mouse. The ability to simulate input by gaze facilitates development of the iComponent core as well as functionality tests of the experiment modules on a standard computer. A converter is active only in the recording mode.

Another set of plug-in modules serves running experiments. In the terminology of iComponent, these modules are called 'experiments'. When an experiment is loaded, it receives all the eye tracker data and events. Moreover, an experiment can send its own events to control iComponent and query its current status, data capture and recording settings.

The experiment modules have a unique interface, organized in the same way as the interface of the converter modules (Figure 2). The modules can be activated in all iComponent modes (one at a time). Evident examples of experiments are on-screen keyboards for text entry via gaze (see, e.g., [5]).

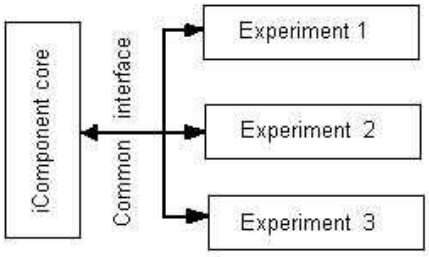

Figure 2. Experiment modules

Currently, the gaze-path visualization modules are built-in, as opposed to DLLs in the case of converters and experiments. However, future versions of iComponent could be implemented with dynamically linkable visualization modules.

Only one visualization scene can be rendered at a time. Visualizations are supported by a set of tools: zooming, navigation, selection, clustering, and replay. These are parts of the iComponent UI. Along with these, there are also a fixation detector and extractor. The detector detects fixations via a selected algorithm with user-adjustable settings and saves them in the fixations file, whereas the extractor prints single fixations one by one to an image file. These additional tools are DLL plug-ins as well.

Overall, the architecture of iComponent leans towards becoming independent of the core functions. Dynamically attachable modules have common interfaces, which makes their implementation easier. Moreover, this helps to avoid recompilation of the executable file.

The full diagram of the iComponent architecture is presented in Figure 3. Besides the converters, experiments, and visualization modules mentioned, a set of managers serve users in reaching their goals. The Quick-Start wizard guides a user through all the steps necessary to complete a task. Study Manager provides functionality for planning and executing experiments: a user may supply experiment settings for use during a recording, and see all the recorded files grouped for further visualization. All the settings and data are profiled. Profile Manager is the first dialog to appear when the application starts. Users may create their own accounts, or choose an existing one for login. Profiles are password-protected.

After installation, iComponent keeps all the related files (API converters and experiment plug-ins, user-related settings and the data recorded) in separate subfolders. The iComponent installation package also includes the devices' API libraries provided by manufacturers.

3. Interfaces

3.1. Device Driver API Converters

The API converters of iComponent are modules (the DLLs residing in the same folder) between the iComponent core and device drivers. Upon starting, iComponent creates a list of supported devices. The user is to verify that the driver to be used by iComponent matches the hardware attached.

There is a template available for an API converter written in C++ to facilitate development of converters for other eye trackers. The converter's API interface consists of the following 15 functions:

```
typedef void (__stdcall FAR*
FiETUDOnSync)(long*, long*);
__stdcall bool Create (HWND
aDataHandler, char* aCallerFolder);
__stdcall void Destroy (VOID);
__stdcall void GetName (char* aName);
__stdcall void ShowOptions (VOID);
__stdcall bool Connect (VOID);
```

```
__stdcall void Disconnect (VOID);
__stdcall bool Calibrate (VOID);
__stdcall bool StopCalibration (VOID);
__stdcall bool CorrectDrift (VOID);
__stdcall bool Start (VOID);
__stdcall void Stop (VOID);
```

```
__stdcall void GetDeviceState
(SiETUDDeviceState* aState);
__stdcall void SetDeviceState
(SiETUDDeviceState* aState);
__stdcall void SetOnSyncEvent
(FiETUDOnSync aCallbackFunc);
__stdcall void PassValue (VOID* aData,
long aType);
```

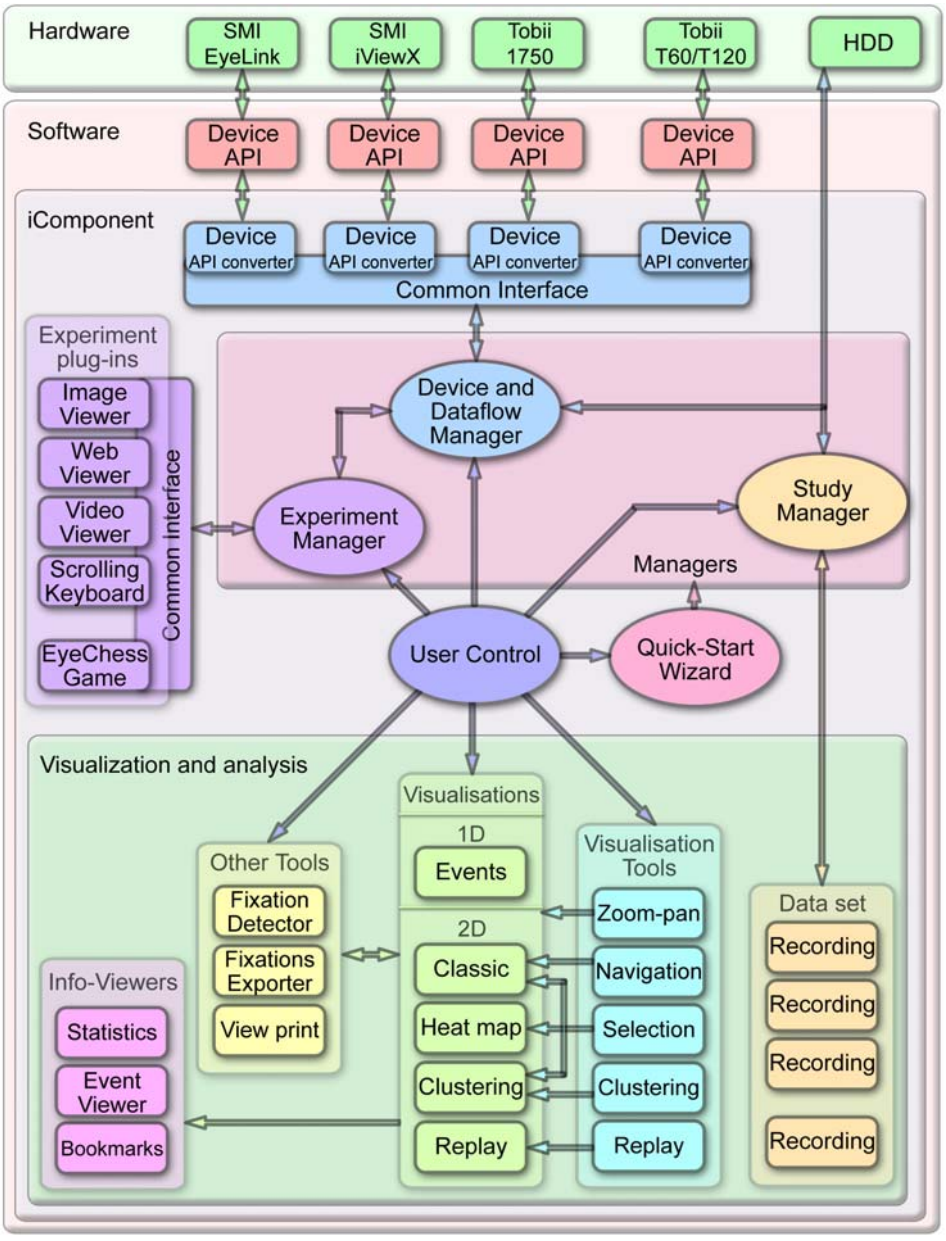

Figure 3. Architecture of iComponent (EyeLink and iViewX are trademarks of SensoMotoric Instruments GmbH; Tobii 1750, T60, and T120 are trademarks of Tobii Technology AB)

Besides creation and destruction of a converter's instance, the API allows setting call-back functions that are executed when new gaze data become available. One API function shows a dialog with settings specific to the device, whereas other two return its name and main parameters. The remaining functions are used to control the eye tracker for these operations:

- connecting to a device and making other preparations so that the device is ready to be calibrated and track an eye (some eye trackers are ready for detecting gaze position as soon as the PC is turned on – then, the function does nothing);
- disconnecting from a device and releasing all the memory used by the connection function;
- starting the calibration routine (this can be implemented either in the manufacturer's device driver, or in its iComponent API converter; in the first case, the function simply calls a similar function

from the device driver), where the calibration usually involves following nine or sixteen targets presented one by one on the screen to form a 3x3 or 4x4 grid;

- starting the drift correction routine (this procedure can be implemented either in the manufacturer's device driver, or in its iComponent API converter), where the drift correction usually involves a single target in the centre of the screen used to make participants hold their gaze on it while the software detects and compensates the drift;
- starting and stopping detection of gaze position, and streaming the data to the iComponent core through the call-back functions mentioned above.

The part of the iComponent core that communicates with device drivers was separated into a standalone COM server so that other developers have an easy way to build gaze-controlled and gaze-aware device-independent applications[1].

The communication interface and its functions described here offer a starting point in development of a standard eye tracking API to allow eye tracking devices to be plug-and-play. However, standardization is impossible without agreement between manufacturers and, probably, operating system developers on common protocols, interfaces, and functions to be used.

3.2. Experiment Plug-Ins

The iComponent 'experiments' are DLL modules having the same functional interface. They reside in the same folder, just like the converters. iComponent detects each experiment and places its name in the 'Experiments' menu on start-up. Each plug-in implements only four functions of the communication with the core interface:

```
__stdcall long ExpParser(MSG aMsg);
__stdcall void ExpGetName(char*
aName);
__stdcall bool ExpShowOptions(VOID);
__stdcall bool ExpIsReady(VOID);
```

The function *ExpParser* plays the most essential role since it receives all the application, device, and gaze events from the iComponent core.

Developers of experiment plug-ins are able to adjust the recording settings and the state of iComponent. For example, it is possible to turn recording of data to file off when not required by the experiment (e.g., eye typing on a virtual keyboard), or to automatically stop the recording session when needed. This can be achieved by sending a particular Windows-style message to the iComponent core that includes a procedure for parsing such messages.

The full list of actions that can be executed in the iComponent core from an experiment plug-in includes the following items: starting and stopping data recording; retrieving and changing recording settings; performing drift correction; toggling full-screen mode on and off; and getting device, profile, and study names.

Other types of messages recognized in the iComponent core consist of system events (mouse click, button press, document scrolling, and so on) and experiment events (image or web page display, or start of video playback) that iComponent saves in the corresponding data file. These events are used later for visualization along with the gaze data.

A typical experiment module consists of at least three parts: 1) implementation of the interface described above; 2) experiment window that uses all the client area in the iComponent main window and presents stimuli during the recording; and 3) dialog with the settings for this module. The basic features of these modules are implemented in a template project written in C++. This template facilitates implementation of new experiment modules. Other developers should be able to use it after having studied the streaming rules for events and data.

3.3. Additional Tools: Fixation Detector

The Fixation Detector tool is the most important one among the additional tools (those that do not support visualizations and gathering of gaze data). This tool is a library for detecting fixations from incoming raw gaze position data. It is implemented as a COM server and thus is easy to embed into any other application that requires fixation detection from raw eye movement data.

Before starting detection, the COM server object must be initialized, to which the application must then send all recording samples from first to last. Finally, the client application informs the server about the end of detection. The server notifies clients during detection about fixation start, update, and end. Developers may write handlers for these events to collect fixations, since the server sends as a parameter the reference to the pointer to data of the detected fixation. Client applications can retrieve the detector's settings as well as modify them.

The COM interface provides seven algorithm-independent settings to be adjusted before fixation detection starts: analyzer type, filter type, buffer size (number of samples to be used in averaging) and weight value, minimum fixation duration (the value below which the fixation is treated as part of a saccade), usage of continuing fixation updating events, and updating interval.

All analysis algorithms can be used for fixation detection in real time as well as in off-line mode. The

[1] This part, namely "Eye-Tracking Universal Driver" (shortly, ETU-Driver tool), was implemented as a COM server that has all the iComponent features related to communication with eye tracking devices. Development of this tool, now having more features and functionality than it had initially being part of iComponent, continues. See http://www.cs.uta.fi/~oleg/etud.html for a more detailed description of the tool.

library can use one of three analyzers to detect fixations:

- The *Fixation Size* analyzer is based on grouping adjacent samples within a circle of certain radius.
- The *Speed* analyzer groups adjacent samples based on speed and acceleration values falling below particular thresholds.
- The *Dispersion* analyzer is based on grouping adjacent samples into a fixation if the dispersion D_i for each sample in the group is less than certain threshold value:

$$D_i = (\max_{i-n..i}(X) - \min_{i-n..i}(X)) + (\max_{i-n..i}(Y) - \min_{i-n..i}(Y)). \quad (1)$$

A data-smoothing filter may be applied to decrease noise before parsing the raw data. One kind of such a filter may be simple averaging over several points:

$$XY_k = \frac{\sum_{i=0}^{n} XY_{k-i}}{n}, \quad (2)$$

where n is the buffer size.

A more sophisticated filter uses weighted averaging, with each sample having its own weight. The last sample ($i = 0$) has a weight $W_0 = 1$; the next-to-last sample ($i = 1$) has a weight W_1 that is equal to the value W defined by the user; the weight of the sample preceding the next-to-last one ($i = 2$) is $W_2 = W^2$, and so on.

In general, $W_i = W^i$. The value of W can be from 0.01 (almost no filtering) to 0.99 (almost as in simple averaging). The weighted filter can be described by the following equation:

$$XY_k = \frac{\sum_{i=0}^{n} XY_{k-i} + (1-W^i)*(XY_k - XY_{k-i})}{n}. \quad (3)$$

To illustrate how this filter modifies the original values, consider the following three sample points ($n = 3$) from raw-data files.

```
X1 = 106    Y1 = 315
X2 = 109    Y2 = 317
X3 = 111    Y3 = 318
```

Setting $W = 0.8$ yields the values of the new position of the next-to-last sample as shown in Equation (4).

$$\overline{X}_3 = \frac{X_3 + (X_2 + (1-0.8)\cdot(X_3 - X_2)) + (X_1 + (1-0.8\cdot 0.8)\cdot(X_3 - X_1))}{3} = \frac{1.56\cdot X_3 + 0.8\cdot X_2 + 0.64\cdot X_1}{3} = 109.4,$$

$$\overline{Y}_3 = \frac{1.56\cdot Y_3 + 0.8\cdot Y_2 + 0.64\cdot Y_1}{3} = 317.1. \quad (4)$$

The data of each fixation detected consist of time, duration, and <*x*, *y*> coordinates. Since the analyzers parse data for a single eye only, two instances of the analyzer must be used to support binocular data. The software that sends data to the analyzer receives the detection result (fixation start, fixation end, and fixation updating events) and is responsible for correct fixation eye labeling and saving this to a file.

The *Fixation Size* analyzer requires a value for maximum distance from the centre of the current fixation to the new sample ('radius of fixation'). The analyzer can reduce the noise effects and filter out individual outlying samples: with the appropriate flag set, the algorithm does not finish the current fixation even if the new samples are at a distance greater than the value of fixation radius. This continues until those samples can be treated as a new fixation (when its duration exceeds the threshold value). If a subsequent sample lies within the current fixation area, all the outlying samples will be considered belonging to the current fixation.

The *Speed* analyzer requires velocity and acceleration thresholds (the maximum values valid for a particular fixation) as well as the buffer size (number of samples to be used in the velocity and acceleration computations).

Finally, the *Dispersion* analyzer requires a dispersion threshold (the maximum dispersion among the fixation's samples) and buffer size (number of samples for calculating dispersion).

4. Generic Data Format

The absence of a common data format, coupled with sophisticated technology (that is not plug-and-play), requires great effort from developers in dealing with gaze data gathered by different eye trackers. Because of lack in standardization, time and money are wasted, and many difficulties arise when using eye trackers in the laboratory settings and everyday life.

To tackle this, we developed a data format that we believe could serve as a universal approach to formatting data in future systems.

4.1. Data Files Topology

The proposed data format was developed after careful study of on-line data generated by various eye tracking systems available to date. The number of eye-movement coordinates provided by eye trackers can be: one (<*x*> or <*y*>), two (<*x*, *y*>), or three: (<*x*, *y*, *z*>, with the latter denoting torsion around the line of sight). A few systems provide pupil size with each data sample. Many eye trackers parse data via detectors of fixations, blinks, and other types of eye movement. Some provide head-movement data as well.

To accommodate various eye trackers, it is reasonable to store the data available on-line in a file's header. The file is then populated with raw data according to the information contained in the header.

All data reside in four files: the main file, fixations file, blinks file, and system and experimental events

file. The main file has a header and a body. The header contains several variables characterizing the recording – general data (such a date and time), personal user data (name, age, etc.), and recording-related data (coordinates recorded by the eye tracker, name of the captured eye, and availability of pupil-size data).

The body of the main file contains raw data (one record per sample). Its structure depends on the information stored in the header including flags to indicate availability of data for each column. Table 1 illustrates the case where all possible types of data are available from the eye tracker.

Table 1. Structure of samples file

Timestamp	Eye									
	Left					Right				
	X	Y	Z	Event	Pupil size	X	Y	Z	Event	Pupil size

The fixations file contains records of each fixation. Each record includes number of the fixation, its start time (time of the first sample in that fixation), duration, identity of the performing eye (left/right), and its coordinates. For binocular tracking, most of the time fixations appear twice (with the same number, but different values in the column *Eye*).

The blinks file contains records of data on individual blinks. Each record contains number of the blink, its start time, duration, and the eye involved. For binocular tracking, blinks usually appear twice, but have a different value in the column *Eye*. Sometimes, however, the eye tracking device or some other supplementary software recognizes a lost-eye event as a blink. Such dummy 'blinks' can be identified by missing pairs.

The system and experimental events file contains both operating-system events and experimental events. Each record in this file includes timestamp of the event, data type, and data themselves. The timer generating timestamps in raw data must be used for timestamps of the events as well. The data are stored in a text format. For instance, in a mouse-click event, a sample string '123,456' contains the coordinates of that click. The software then recognizes that this kind of event has *x* and *y* values separated by a comma.

System events are mouse clicks, keystrokes, and scrolling of stimuli. Experimental events are those related to execution of the experiment and presentation of stimuli. The most common stimuli are images, web pages, and videos. For images and videos, the data string contains the file path; for a web page, its URL is stored along with the number of the record.

For a recording to be considered valid, the main file with raw data suffices, since both the fixations and blinks files can be generated from raw data via fixation and blink detectors.

4.2. Data Units

Every record of a sample starts with the time in milliseconds relative to the start time of the recording. It is better to use time measured by the device than the PC clock time that may drift depending on the machine's state. However, not all devices provide timestamps on request, so special software must be used for correct timing.

Coordinates are real numbers from 0 to 1 for a gaze landing on the PC screen. The 'event' column contains 32-bit integer values to indicate the type of eye movement that occurred during sampling of gaze direction. iComponent currently supports two basic types of eye movement: fixations and saccades. For samples belonging to a saccade, the flag is set to 0. Meanwhile, positive values indicate samples belonging to a fixation.

The current version of the software does not accommodate other types of eye movement (such as smooth pursuit, nystagmus, or vergence). However, several of the higher-level bits are reserved for future use. Moreover, the two highest bits indicate quality of the sample obtained; large negative values of this flag purport a sample of poor quality. Data on the validity of the samples were included to support several newer eye trackers providing this kind of information.

Although the data format proposed supports collection of pupil-size values, these depend on the type of camera, image processing algorithm, and distance from the eye. Thus, it may be impossible to convert them into the actual pupil size (expressed in square millimeters, for instance). Different eye trackers provide a very wide range of values (1–100,000) and report different parameters (the most common parameter, however, is the number of pixels covered by the pupil). Consequently, only analysis of pupil dilation might be meaningful to researchers.

4.3. Data Example

Suppose that we have a common case where raw data comes from a VOG-based eye tracker measuring *x* and *y* coordinates as well as pupil size for both eyes. Then, the main file's body will be formatted as shown in Table 2.

In our hypothetical example with sampling rate of 250 Hz, the first four samples belong to a saccade (as indicated by zeroes in both the event columns) due to a blink (as revealed by the gap in time and the event's value of –1 in the following sample). In addition, a very large value in column *LEvent* indicates probable invalidity of the left eye's data. Finally, the last three samples in Table 2 belong to fixation number 1.

Table 2. Example of raw data

Time	LX	LY	LEvent	LPupil	RX	RY	REvent	RPupil
0	170	55	0	1240	190	52	0	1265
4	181	56	0	1245	198	52	0	1265
8	192	58	0	1248	205	53	0	1266
12	211	-1156	1342177280	785	210	40	0	1266
100	219	23	-1	1125	227	20	-1	1178
104	217	25	0	1129	126	21	0	1181
104	217	25	1	1128	127	21	1	1182
108	217	26	1	1127	127	23	1	1183
112	217	26	1	1128	126	23	1	1181

5. Conclusions

Architecture of iComponent allows easy development and use of new device-supporting modules and experiment plug-ins. Some portions of iComponent are already implemented in separate tools (such as ETU-Driver and Fixation Detector). Further modularization of the iComponent core into separate libraries (e.g., plug-ins for visualizations) should bring yet more flexibility to the software.

The generic format developed allows storing all relevant gaze data along with the events related to the environment, stimuli, and human activity. The current version of the format does not support all possible data available from some eye trackers, though. For example, the resolution of timestamps is only at the milliseconds level. Therefore, the format suits eye trackers having sampling frequency of 1000 Hz at most. However, this limitation should be diminished in the future.

Hopefully, our work described in this paper will serve as a starting point for standardization of eye tracking devices. Moreover, the proposal has been presented as official recommendations for manufacturers of eye tracking devices [1].

References

[1] **R. Bates, H. Istance, O. Spakov.** D2.2 Requirements for the Common Format of Eye Movement Data. *Communication by Gaze Interaction (COGAIN), IST-*2003-511598*: Deliverable* 2.2, 2005. Available at *<http://www.cogain.org/results/reports/COGAIN-D2.2.pdf>*.

[2] **A. Duchowski.** A breadth-first survey of eye-tracking applications. *Behavior Research Methods, Instruments & Computers* 34, 2002, 455-470.

[3] **J.P. Hansen, D.W. Hansen, A.S. Johansen.** Bringing gaze-based interaction back to basics. *C. Stephanidis (ed.), Universal Access in HCI, Lawrence Erlbaum Associates*, 2001, 325–328.

[4] **R.J. K. Jacob, K.S. Karn.** Eye tracking in human-computer interaction and usability research: ready to deliver the promises. *J. Hyönä, R. Radach, and H. Deubel (eds.), The Mind's eye: Cognitive and Applied Aspects of Eye Movement Research, Elsevier Science*, 2003, 573-605.

[5] **D. Miniotas, O. Špakov, G. Evreinov.** A comparative study on two eye-based text entry techniques. *Information Technology and Control* 27, *Kaunas, Technologija*, 2003, 63-68.

[6] **C. Morimoto, M. Mimica.** Eye gaze tracking techniques for interactive applications. *Computer Vision and Image Understanding* 98, 2005, 4-24.

[7] **R. Ramloll, C. Trepagnier, M. Sebrechts, J. Beedasy.** Gaze data visualization tools: opportunities and challenges. *Proc. IEEE Conference on Information Visualization*, 2004, 173–180.

Received January 2007.