Highly Configurable Software Architecture Framework for Acquisition and Visualization of Biometric Data

Jan Stelovsky

University of Hawaii, Department of Information and Computer Sciences
1680 East-West Road, Honolulu, Hawaii 96816, USA

Abstract. The research in augmented cognition and its practical applications rely heavily on the acquisition and evaluation of biometrics data. We propose software architecture that offers unified approach to the integration of emerging hardware and evaluation technologies. In this paper we focus on the software layers that combine the data events and offer visual representations of the results. In particular, we show that the common evaluation of the collected data as well as the commonly used graphical depictions of the results can be achieved using a fully modular and extendible software architecture.

Keywords: software architecture, visualization, biometrics, eye-tracking.

1 Introduction

Acquisition of biometrics data, its evaluation and visualization of the results are key components of the research on human cognition. One of the main reasons that after years of basic and applied research the current technology is still far from being integrated into practical applications is that experiments are very time-consuming to design, administer and evaluate. While new biometric sensors are being developed and new visualization techniques are proposed, the software environment needed to analyze the data is typically highly sensor-dependent and hard-coded. Moreover, the visualizations tend to be complex in order to depict the experimental data in various perspectives and thus less suitable in practical applications where simpler user interfaces are customary, e.g. a set of gauges that aggregate several data dimensions.

To simplify the task of integrating all aspects of data collection, processing, and visualization for experimental purposes, we have been developing EventStream software framework that satisfies the following objectives:

- Incorporates various sensors, e.g., equipment measuring temperature, heart rate and skin resistance, pressure on the mouse case and buttons, eye-tracker, oximeter, interactive events (keyboard events, mouse movement and clicks).
- Supports various data sources and sinks, such as file, LAN and Internet (TCP/IP) streams, serial ports, interactive and software-generated events.
- Incorporates common data evaluation and display methodologies.
- Is based on unified software architecture that offers higher level abstractions, atomic building blocks as well as commonly used composite functionality.
- Is highly extendible with Java modules that can be combined and reconfigured.

D.D. Schmorrow, L.M. Reeves (Eds.): Augmented Cognition, HCII 2007, LNAI 4565, pp. 176–185, 2007.

The software architecture that defined the input and output functionality, i.e. the support for the existing and emerging sensors and data sinks was described in detail in [1] and [2]. One of the challenges was to unify the real-time visualization of incoming data from interactive events with the replay of the experimental session that used data read from a data stream. Instead of converting this particular type of data into events or developing two display modules, we developed a generic solution – a set of conversion modules that can convert arbitrary events into data that can be deposited onto a stream or send data retrieved from a stream to listeners in the form of events. The time intervals in between two consecutive events were proportional to the time span originally recorded and the scaling factor could be freely chosen. The design and implementation of this facility was described in [3]. The data processing code and the implementation of various graphical representations of the results remained, however, hardwired and tailored specifically to each case.

In this article we shall first analyze the common visualization techniques and then propose a software architecture that extends the unification and extensibility effort to the data processing and visualization modules and describe a prototype implementation of the architecture's core elements.

2 Common Graphical Visualizations of Experimental Data

The evaluation and visualization of eye tracking data is one of the more complex endeavors in biometric experiments because of the magnitude of the data obtained as well as its referential complexity. Let us therefore look at some implemented and proposed graphical representations as they were collected and presented in [4].

A common presentation of event-based data is a 2-dimensional graph where one of the axes represents the elapsed time. While measurement data forms a discrete function, a polyline connecting the samples is an adequate visualization. For example, the 2-dimensional graph in Fig. 1 shows the pupil size during an eye-tracking session.

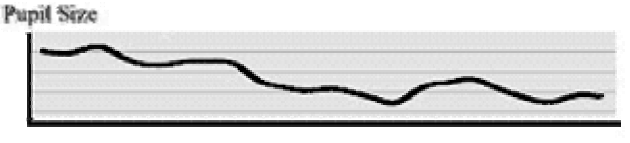

Fig. 1. A 2-dimensional graph depicting the *pupil size* with respect to *time axis*

The raw eye tracking data has to be analyzed in terms of fixations – numerous eye gaze points falling into the same area – and saccades – the rapid eye movements. Even though there is general agreement that visual information is processed only during the fixations, graphs that combine saccades and fixations are common in literature as they convey the impression of a continuous gaze path. Such combined Fixation Path graph is depicted in Fig. 2. The dynamic version of Fixation Path graph where the sequence of fixations is shown as a (possibly slowed down) movie is also very useful as it indicates where and for how long the subject looked.

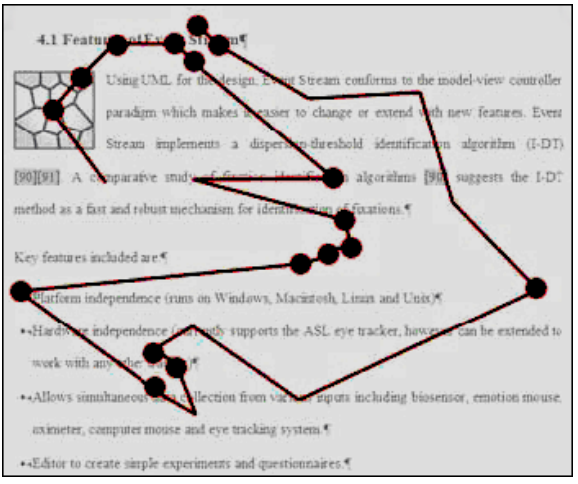

Fig. 2. A Fixation Path graph with *fixations* (*black circles*) and *saccades* (*lines*) on a *background image* that shows the text presented to the subject

The Transition Matrix graph in Fig. 3 identifies areas of interest, determines the probabilities of a transition between consecutive fixations in different areas. These probabilities are then depicted as oriented arrows between the areas of interest labeled with the percentage numeric values corresponding to the probabilities.

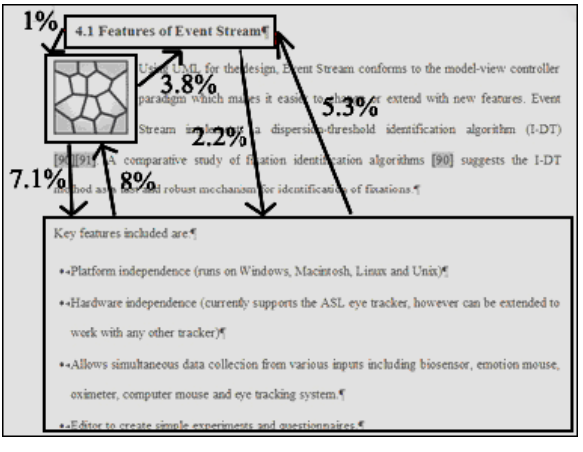

Fig. 3. A Transition Matrix graph with *areas of interest (rectangles)* and *transition probabilities (arrows with percentages)* superimposed on a *background image*

Another often used visualization is the Heatmap graph shown in Fig. 4. It depicts the overall hotspots of fixations as ovals whose color and size indicate the frequency of fixations within the underlying area.

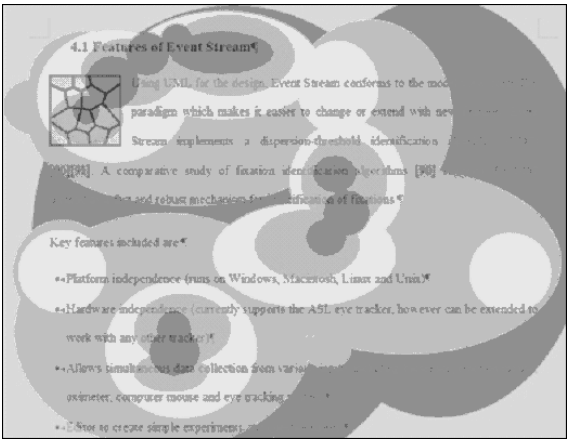

Fig. 4. A Heatmap graph with *fixations hotspots (colored ovals)* on a *background image*

Other graphical representations of biometric data are commonly used, among them those that use surfaces in 3-dimensional space, which are just extensions of curves within a 2-dimensional graph into 3-dimensional space in our context.

The above discussed visualizations were not selected according to their usefulness for the analysis of eye movement data. Instead, we tried to choose a wide spectrum of visually different representations.

3 Function Framework

Before the biometric data can be visualized, it needs to preprocessed. For instance, raw eye gaze data must be condensed into fixations. For the purposes of augmented cognition experiments, several dimensions of the data are typically combined to obtain indication of the subject's mental state and cognitive capabilities. For instance, higher heart rate and increased pressure on the mouse case may indicate stress levels that impede cognitive abilities. The schema in Fig. 5 shows how a function $F(x_1,\ldots,x_n)$ can aggregate the data vector into one value which is then shown on a gauge.

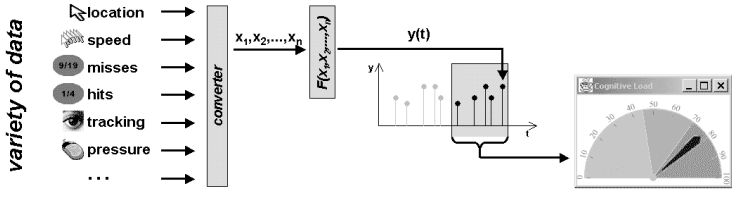

Fig. 5. Schematic data flow coalescing in a *cognitive gauge*

A more general schema may aggregate data over a recent time span in individual dimensions before combining them or employ several gauges to show different

aspects of the cognitive state. In particular, a framework should allow the experimenter can choose the functions arbitrarily.

The biometric data consists of time-stamped events that form a discrete function in multidimensional space. This function is a series of samples of a continuous function that can be approximated by interconnecting the sampled values. In general, the preprocessing can be regarded as a function from one multidimensional space into another. We have therefore designed an extensible function framework that allows for definition of functional building blocks that can be combined into complex composite functions. The UML class diagram in Fig. 6 depicts the structure of the framework.

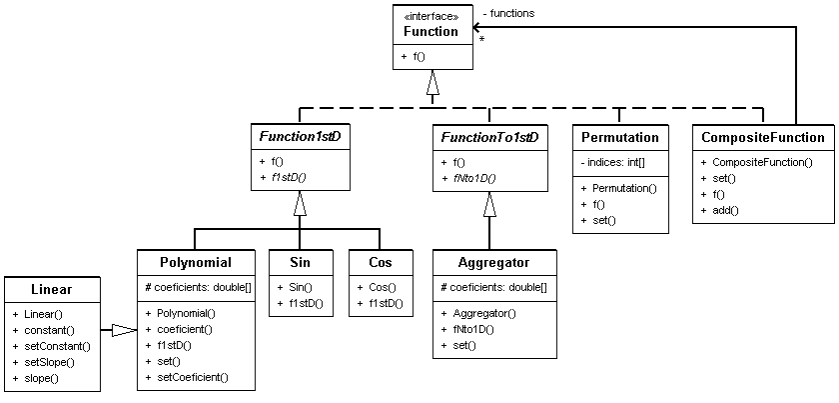

Fig. 6. UML class diagram of the function framework

At the heart of the framework is an interface contract that prescribes that a function must return a multidimensional point f(x) given a multidimensional point x. (The spaces may have different dimensions.)

```
public interface Function {
  public Point f (Point x);
}
```

We define a special case for one-dimensional functions. To be able to treat them the same way as the generic multidimensional functions we postulate that they change only the first coordinate of the given point and leave the other dimensions untouched.

```
public abstract class Function1stD implements Function {

  public abstract double f1stD (double x);

  public Point f (Point x) {
    x.vector [0] = f1stD (x.vector [0]);
    return x;
  }
}
```

To give an example, let us demonstrate the implementation of a particularly common one-dimensional function – the polynomial function.

```
public class Polynomial extends Function1stD {
  protected double coeficients [];
  public double f1stD (double x) {
    double value = 0;
    for (int i = coeficients.length - 1; i >= 0; i--) {
      value = value * x + coeficients [i];
    }
    return value;
  }
}
```

The implementation of a `Linear` function is trivial as a subclass of `Polynomial` with a constructor that has the two parameters `slope` and `constant`.

Finally, let us demonstrate the implementation of function composition. (We present a simplified version without the accessor methods for the properties.)

```
public class CompositeFunction implements Function {
  private ArrayList<Function> functions;
  public CompositeFunction (Function... functions) {
    for (Function function : functions) {
      this.functions.add (function);
    }
  }
  public Point f (Point x) {
    for (Function g : functions) {x = g.f (x);}
    return x;
  }
}
```

We support other useful functions as well. For instance, we define the basis for aggregate functions that can combine the values in several dimensions. Functions that filter the data or reduce the number of dimensions are trivial extensions of this framework. (The class `Point` itself provides for the composition of two points, one N-dimensional and the other M-dimensional, into a point in N+M-dimensional space.)

The architecture presented above provides for an extremely simple, yet powerful framework. Together with the standard mathematical functions, this framework can accommodate a large variety of data-processing functions. As we will see later, graphical visualization methods can, for instance, use a composition of `Linear` functions with a function that permutates the coordinates of a point to scale the data to fit the drawing specifications and screen constraints. Obviously, the applications of the functional framework extend beyond the evaluation and visual representation of data as the framework provides a basis amenable to further mathematical treatment.

4 Visualization Framework

Closer analysis of the graphs in Fig. 1 through Fig. 4 reveals that these visualizations have a common basis – there is some "background graphics" such as graph axes, an image, or potentially continuous media such as sound or video. Superimposed on this media is some representation of points: as a sequence of line segments (Fig. 1 and Fig. 2), circles (Fig. 2), rectangles (Fig. 3), ovals (Fig. 4), arrows (Fig. 3), or textual labels (numbers in Fig. 3). Every Java programmer will associate these geometrical shapes immediately with the methods for drawing primitive shapes in her preferred graphical support package. Furthermore, we observe that the shapes tend to have more parameters than the x and y coordinates, e.g., the size and color of the ovals in the heatmap in Fig. 4 or the numeric values in Fig. 3. Also, each of the lines or arrows needs an additional point, i.e. another pair of x and y coordinates. Moreover, some graphs combine several representations, such as lines and circles in Fig. 2. Despite these complexities the implementation of the graphs is fairly straightforward.

Can the implementation of such visualizations be facilitated even more by a common underlying framework? Probably every programmer will write common software modules for drawing the axes and scaling the x and y coordinates. But further commonalities emerge if we treat a visualization as the representation of a sequence of points in a multidimensional space. (It is of course no coincidence that the function framework discussed in the previous section provides us exactly with such points.) The responsibility of a graph is therefore to provide such a sequence of points. Then each of the points can be drawn by a specific "painter". (Notice that the "background graphics" in the last paragraph can be also interpreted as one of the coordinates – then a frames number of a continuous media can be represented as a coordinate value and the painter will paint the corresponding frame.) The UML class diagram in Fig. 7 shows the basic structure of the proposed visualization framework.

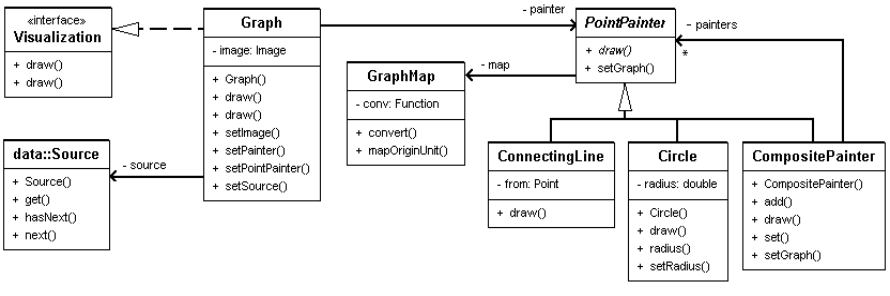

Fig. 7. UML class diagram representation of the graphical visualization framework

The core of the framework is the `Graph` class that supplies one `draw()` method that draws the entire sequence of points and another `draw()` method that the paints the "current" point. It requests the points from the `Source` of data and uses a `PointPainter` to draw each of them. Again, let us illustrate this with a simplified version of the code:

```
public class Graph implements Visualization {

  private Source source;
  private PointPainter painter;

  public Graph (Source source) {this.source = source;}

  public void setPointPainter (PointPainter painter) {
    this.painter = painter;
    painter.setMap (new GraphMap (new Point(0,100)
        , new Point(10,0));
  }

  public void draw (Graphics graphics) {
    while (source.hasNext ()) {
      painter.draw (graphics, source.next ().get ());
    }
  }

  public void draw (Graphics graphics, Point point) {
    painter.draw (graphics, point);
  }
}
```

Notice that the while the `source` is responsible for preparing the `Point` data, the `painter` can convert it to the actual screen coordinates using the class `GraphMap`:

```
public class GraphMap {

  private Function conversion;

  public void mapOriginUnit (Point o, Point u) {
    Linear lx = new Linear (o.x(), u.x() - o.x());
    Linear ly = new Linear (o.y(), u.y() - o.y());
    Permutation pm = new Permutation (0, 1);
    conversion = new CompositeFunction (lx, pm, ly, pm);

  }

  public Point convert (Point point) {
    return conversion.f (new Point (point));
  }
}
```

The convenience method `mapOriginUnit()` illustrates how this framework can be used to quickly prototype the scaling and transition needed to map a data point to the actual position on a screen pane – otherwise a quite error-prone operation due to the fact that the y axis on the screen runs in the opposite direction than we are used to see in a book. Suppose that the origin and unit points contain the screen coordinates of the (0,0) and (1,1) points in the data space. Then the conversion to the screen point is given by two linear functions as defined in the above code. We can use a composition of these functions if we interchange the coordinates before we apply the second linear function and interchange them again at the end. The application of this composite function within the `convert()` method then becomes trivial.

Needless to say, this code is not a masterpiece of efficiency as four functions will be applied every time a point is drawn. On the other hand, the overhead will be typically negligible compared to the effort needed to draw the pixels of a line or an oval, and the clarity of the code (once the coordinate interchanges become second nature) is a clear advantage. Moreover, the above code can be easily modified into a more efficient version.

Finally, as an example of a `PointPainter`, let us present a simplified version of a class that draws a circle:

```
public class Circle extends PointPainter {

  private int radius = 10;
  private GraphMap map;

  public Circle (int radius) {this.radius = radius;}

  public void setMap (GraphMap map) {this.map = map;}

  public void draw (Graphics graphics, Point point) {
    Point p = map.convert (data);
    graphics.gc ().fillOval (Graph.round(p.x()-radius),
        Graph.round(p.y()-radius), 2*radius, 2*radius);
  }
}
```

Note that instead of letting the painter convert the data into screen coordinates, our function framework would allow us to incorporate the conversion functions into the data-processing by appending them to the end of the function composition. While we opted to leave them as a responsibility of the graph's functionality, we could also let the graph pass them to the source for integration.

A careful reader may ask: What if our circles needed to vary their radius and color to reflect additional dimensions of the source data? Then the source will have to add other two dimensions to our points and the painter will need to convert these coordinates into the appropriate values of the corresponding data types. This is simple with numeric data such as the circle radius and can be incorporated into the conversion within the `GraphMap` class. Similarly, a grayscale value can be accommodated within the point's conversion. One could also imagine converting data into the RGB value for color definition or even introducing three separate dimensions for either the proportion of primary colors or – probably more suitably for visualization purposes – the hue, saturation and brightness components of a color. The more general solution, however, is to supply separate conversion functions that map the data points into the appropriate data type.

The visualization framework also incorporates a composite pattern to accommodate representations where one point is visualized in multiple fashions. The implementation of the pattern in `CompositePainter` is almost identical to that of `CompositeFunction`. To give an example, the saccades and fixations in the Fixation Path graph shown in Fig. 2 can be drawn using the composition of a `ConnectingLine` and `Circle` painters.

4 Conclusion

The functional and visualization frameworks present a simple solution that unifies a wide variety of commonly used data processing functions and visualizations. Both frameworks are very extendible. New data processing functions can be constructed either as a composition of existing ones or, if this is not possible, by writing a `Function` subclass in Java. Similarly, complex visualizations can be composed from basic shapes and new painters can be implemented as Java subclasses. Moreover, specifications of function parameters and function compositions can be provided separately from Java code – e.g. in the form of XML configuration files – and the corresponding classes can be dynamically loaded. Parameterization and composition can be even made part of the user interface thus allowing the experimenter to choose the best preprocessing and visual representation of the data.

There is an additional benefit of our approach: the visualization in real time and the replay of an experimental session is a natural extension that does not require any coding effort. Whether the data is generated as a sequence of events in real time or whether it is read from a stream, our EventStream software can provide it to the data processing layer in either fashion. And since the graph drawing layer does not depend on whether the points arrive with or without delay, a replay is generated simply by pausing by a given time span before prompting the graph to draw the next point. Even if the graph insist on getting the consecutive points "immediately" – i.e. within a while loop – the data source can block the drawing thread for the necessary span of time. Note however that this replay facility works only in the forward fashion and more elaborate state "undo/memento" facility would be needed to go backward in time without having to reconstruct the entire session.

A prototype of the proposed frameworks has been implemented in Java using Eclipse and its SWT toolkit. We plan to incorporate an enhanced version into the software visualization project that we are currently developing.

Acknowledgments. Fig. 1 through Fig. 4 courtesy of C. Aschwanden.

References

1. Stelovsky, J., Aschwanden, C.: Measuring Cognitive Load with EventStream Software Framework. In: Sprague, R. (ed.) Proceedings of the 36th Hawaii International Conference of System Sciences (CD-ROM), IEEE Computer Society, Washington (2003)
2. Stelovsky, J.: An Extendible Architecture for the Integration of Eye Tracking and User Interaction Events of Java Programs with Complex User Interfaces. In: Bullinger, H.-J., Ziegler, J. (eds.) Human-Computer Interaction: Ergonomics and User Interfaces, vol. 1, pp. 861–865. Lawrence Erlbaum Associates, Mahwah (1999)
3. Stelovsky, J., Aschwanden, C.: Software Architecture for Unified Management of Event Notification and Stream I/O and its Use for Recording and Analysis of User Events. In: Proceedings of Hawaii International Conference on System Sciences, p. 138 (2002)
4. Aschwanden, C.: EventStream Experimenter Workbench: From Data Collection to Integrated Visualization Techniques to Facilitate Eye Movement Research. Ph.D. Dissertation, University of Hawaii at Manoa (2005)