Managing and displaying user track data with Python

Walter Aprile w.aprile@sssup.it
Emanuele Ruffaldi e.ruffaldi@sssup.it
Antonio Frisoli antony@sssup.it
Massimo Bergamasco m.bergamasco@ssup.it

Scuola Superiore Sant'Anna, Laboratorio PERCRO
Viale R. Piaggio 34, 56025 Pontedera (PI), Italy +39 050 883057

Abstract: User studies that feature user movement in the real world or in simulated environment generate datasets, usually in the form of logfiles, that need to be stored, summarized, processed and represented. Datasets must additionally include metadata that accounts for experimental conditions. We have developed a class that produces graphical displays of user travels over a regularly-spaced grid, and a set of web-controllable database management tools that allow incremental data exploration as the user experiments progress.

Making sense of user movement and user commands

In our research work at the PERceptual RObotics (PERCRO) laboratory of Scuola Superiore Sant'Anna we frequently work with data that describes the movement of users when performing tasks. This data can describe various types of phenomena such as:

1- end effector user-controlled movement of haptic interfaces: a user holds a haptic interface such as a force feedback joystick or a GRAB interface (Avizzano et al., 2003), and we are interested in the movement of the point where the users' fingers contact the physical interface.
2- user navigation through a virtual environment
3- hand and head motion acquired through various optical tracking techniques
4- multi-track movement data representing various points on the user body

The user-generated stream of positions that represent his/her movement(s) is interspersed with additional events that represent variously

1- user commands given through controller devices
2- salient moments in the interaction, such as operational errors, the attainment of certain goals or the existence of certain conditions

The focus of the experimenter can be to measure changes in user behavior and efficiency under different experimental condition, with an eye to optimizing interfaces; or recording user movements and gestures for animating avatars or storing specialized forms of manual knowledge.

The form of user track data

A typical raw data stream from a user navigation experiment could take this form:

```
1171203027.3725 R 95.12       10 95.01    -0.864507 0.0 0.50262
1171203027.3725 UP
1171203028.3765 CAMERA_MAX_VELOCITY_EXCEEDED_FORWARD
1171203028.3765 R 94.81       10 94.97    -0.864507 0.0 0.50262
```

This describes events taking place between 3.725 and 3.765 seconds after the start of the experiment. The user was on a location coded as "R". His coordinates changed from 95.12,10,95.01 to 94.81,10,94.97. His heading (the last three number of the row) stayed constant. At 3.725 seconds he gave the UP command. At 3.765 he reached the maximum velocity allowed in this experiment.
This is is just an example of a possible data format. Other experiments will likely produce (and have produced) different looking data, but they all fall in the same pattern: a sequence of time-coded status snapshots, potentially interspersed with commands and events.
Additionally GPS receivers produce their own mandated format of track data, the NMEA 0183 standard ([NMEA 1994]). To allow comparison and integration of different experiments, our approach is to write an input filter that reads in user track data and converts it into a uniform internal representation. The format intentionally avoids metadata such as experimental conditions, absolute

time of experiment and subject id. The place for such metadata is inside the session database, not in the logfile.

Graphical representation of user track data

One simplistic way of showing user track data is to simply plot the user track on as plane as a x(t),y(t) function of time, possibly augmenting the display with graphics related to the user task.

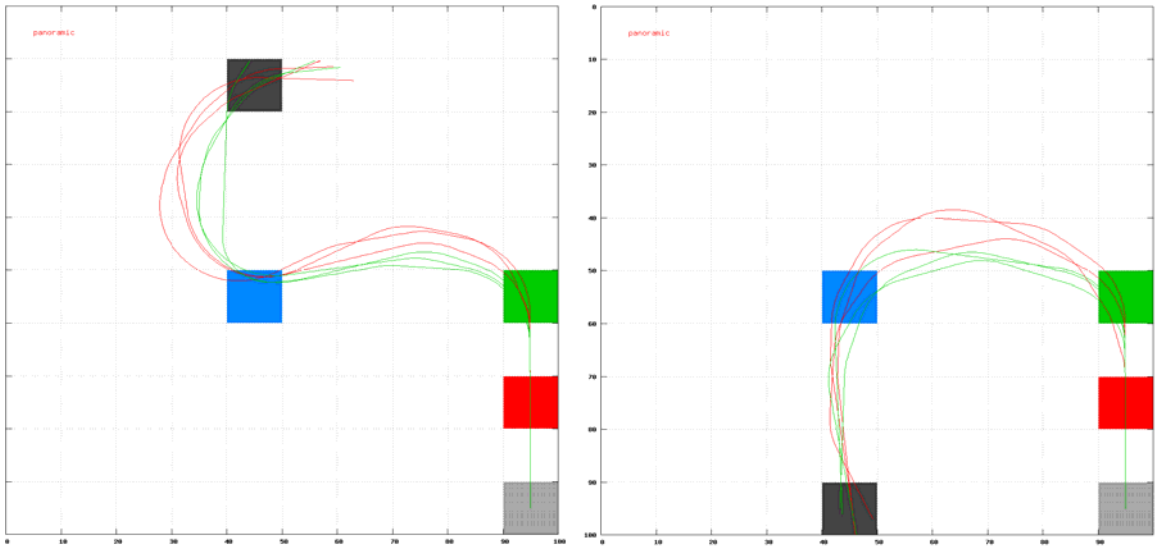

Figure 1: user navigation as x(t),y(t): three trials under two different experimental conditions on two maps. The user starts on the grey square. Task consists of stepping on the red, green, blue and black tiles.

But this type of display is not adequate for large amounts of data. Beyond a certain number of lines it turns into an unreadable knot, and visual patterns become lost in the confusion. A well known technique for summarizing data is *bucketing*, a technique that in its simplest form produces the histogram visualization. A type of 2D bucketing that records how much of the user track intersects a given rectangular element of a grid overlaid on the plane is the *visit map*. A tool for drawing visit maps has been recently proposed in [Chittaro and Ieronutti 2005] and [Ieronutti, Ranon and Chittaro 2005]. The same work describes in detail the use of visit maps in integration with other types of visualization for evaluating user behavior in virtual environments.

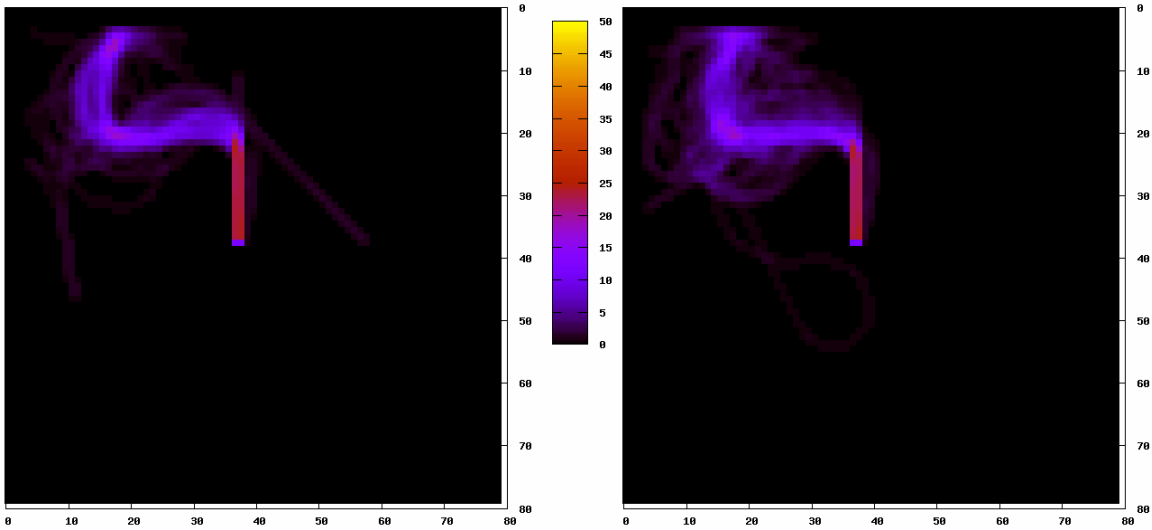

Figure 2: aggregate visit maps of the same task under different experimental conditions

The visit maps in Figure 2 aggregate respectively 30 user navigation sessions on the left map of Figure 1 under two different experimental conditions The trajectories summarized in the map on the left are

visibly more curved, or at least visibly enough to prompt a serious statistical study of the difference in user behavior caused by the changed experimental condition.
A visit map can show interesting patterns also in single experimental sessions.

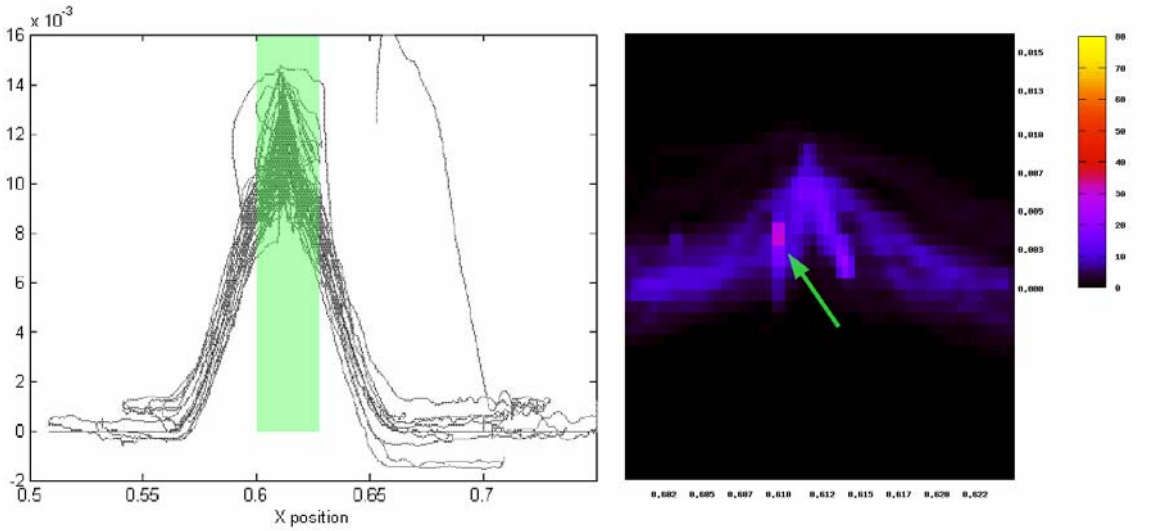

Figure 3: x(t),y(t) representation of interaction with a simulated profile from [Portillo Rodriguez et al. 2006]. Area in green is magnified in visitmap on the right.

Figure 3 shows data from a study of user interacting via a haptic interface with a simulated sharp profile (Portillo Rodriguez et al. 2006). The visit map shows a hotspot, marked with a green arrow, where the user has spent twice as much time as in the surrounding areas. This preference for a particular point in the profile, being a temporal information, is not visible in the x(t), y(t) representation.

Generating density maps from movement logs using a raycasting algorithm to drive gnuplot

We were not able to find a free package that would draw visitmaps to our satisfaction, so we decided to write our own using Python and the freely-available data- and function-plotting program gnuplot (4.0). This was also an exercise in test driven development, as we had the impression that this would be an algorithm very easy to get subtly wrong. Our algorithmic problem consists of marking the cells that are intersected by the segments that make up the user path. The marking can be either a simple boolean, or a float that is proportional to the length of the segment that intersects the cell.

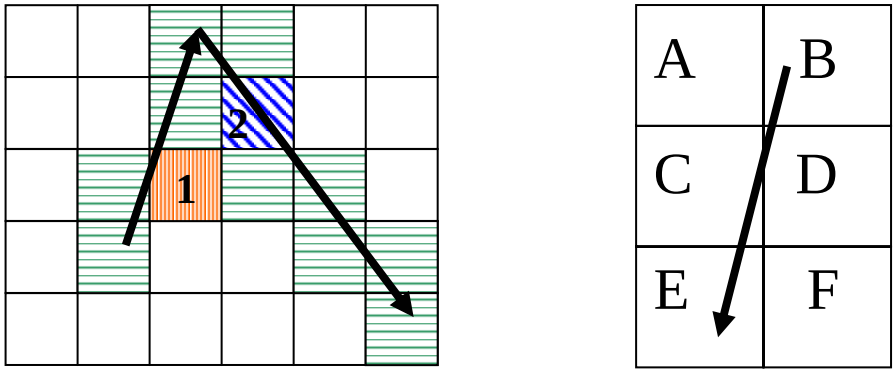

Figure 4: cells that have to be marked, and a case where an antialiasing algorithm will not work.

For example, in figure 3, cell 1 should be marked much less that cell 2, since the intersection is much shorter. At first glance, this problem has strong analogies with antialiased line drawing [Wu 1991]. But a little reflection will suffice to show that in cases like the right part of figure 3, a grayscale line

drawing algorithm will mark all the cells in some measure, while our requirement is that only B C D E be marked.
At the same time, we are interested not only in which cells were touched but also in the list of the intersection points of the line with the cell boundaries.

Though some terminology blurriness exists, *raycasting*[1] (as opposed to *raytracing*) is a fast technique to render a fairly constrained 2D plus elevation map into 3D. Gamers may remember that it was put to great use in the popular Wolfenstein 3D title. Raycasting, like Bresenham's classic line drawing algorithm [Bresenham 1965], is a type of DDA (Digital Differential Analyzer), that's to say a way of reducing a problem that involves floating point computations, in our case incrementing a y coordinate based on the slope of a line, to a problem involving integers.

Our minor modifications to raycasting were necessary because we are interested not only in the cells that are touched by the segment (or ray) we are casting, but also in the length of the intersecting segments.

Our algorithm takes as input two points (x1,y1) and (x2,y2). We provide for trivial cases when the two points fall in the same cell or when the points coincide. Then we compute some constants:

```
rayDirX = x2-x1
rayDirY = y2-y1
m = rayDirY/rayDirX
m1 = rayDirX/rayDirY
q = y1-m*x1
deltaDistX = (1 + (rayDirY * rayDirY) / (rayDirX * rayDirX))**.5
deltaDistY = (1 + (rayDirX * rayDirX) / (rayDirY * rayDirY))**.5
```

We also compute sideDistX, side DistY, deltaDisX and deltaDisY: sideDistX and sideDistY are initially the distance the ray has to travel from it's start position to the first x-side and the first y-side. Later in the code their meaning will slightly change.

deltaDistX and deltaDistY are respectively the distance the ray has to travel to go from one horizontal gridline to the next one, or from a vertical gridline to the next vertical gridline.

Based on the sign of rayDirX and rayDirY, we determine the quadrant that the segment lies in – in other words whether it goes up or down and left or right. This information is stored in stepX and stepY variables. Until we hit the cell that contains x2,y2 we repeat this loop:

```
while (hit == 0):
#jump to next map square, OR in x-direction,
OR in y-direction
    if (sideDistX < sideDistY):
          #hit the vertical side of the
square
          sideDistX += deltaDistX
          mapX += stepX
          side = 0


    else:
          #hit the horizontal side of the square
          sideDistY += deltaDistY
          mapY += stepY
          side = 1
```

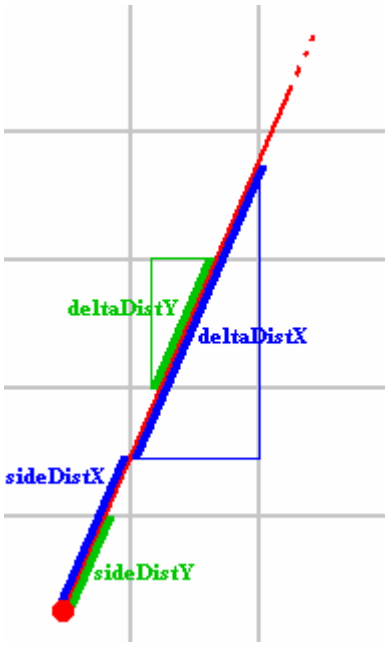

Figure 5. A ray crossing cells.

[1] You can find a very clear raycasting tutorial by Lode Vandevenne at http://student.kuleuven.be/~m0216922/CG/raycasting.html - our algorithm is basically an extension of the one described in detail on this page, from where we have also taken Figure 5.

We also compute the coordinates of the intersections with the m and q line constants that we showed at the beginning. The algorithm returns a data structure that is a list of records containing the side of the square that has been intersected, the coordinates of the point of intersection, the coordinates of the cell and finally the length of the intersecting segment.

For our purposes, we need only the last two elements that identify the map cell and the amount of time that the user has spent inside it.

Figure 5 is Lode Vandevenne's tutorial at http://student.kuleuven.be/~m0216922/CG/raycasting.html

Edge cases and test-driven coding

This type of algorithm is notoriously difficult to get right due to edge cases and fencepost errors. Having suffered in the past from painful development experiences, we decided to try test-driven coding. The algorithm has been wrapped in a visitMap class that is able to record visits to segments and also, trivially, to specific map cells. The class also has methods for dumping its contents in various forms.
All access to the map content (currently stored as a very naïve python list) is done via two get and set methods, with the idea to allow future improvements via specialized representations for sparse matrices.
We wrote a set of test cases that exercise the class methods in order of abstraction, insisting on unpleasant cases like horizontal, vertical and pi/4 slope segments, very short segments, almost horizontal segments and segments that end and/or terminate on integer coordinates that coincide with cell boundaries.
Some of our cases are designed to raise errors, and a longer-running one attempts to draw a large number of pseudorandom segments and then checks whether the total weight of the map is the same as the accumulated length of the segments.
We are happy to report that insistence on testing has proved very advantageous in our work. We feel more confident in our code, and we feel much more relaxed in modifying it, knowing that at any moment we can re-run the test suite and validate the class.

A TurboGears application for managing user movement datasets

The class we have described in the previous paragraph has been developed and used immediately in an ad-hoc harness that picked logfiles out of a directory, parsed them into segments, fed them to the class and then outputted the data in a matrix format that was processed into graphics by gnuplot's pm3d mode. The most irritating part of the process was managing the data. We realized that we wanted to:

- store metadata about the user track data, such as experiment type, experimental conditions, time, location
- store summary data such as task success/failure, total task time, subtask times, mean velocity
- filter the track data before feeding it into a map based on the metadata
- experiment with additional visualization techniques and enable interactive data exploration

These are common needs. There should be a common solution. This is why we decided to build a database-backed web application for managing the data.

Data model

The central object/table of our application is the experimental session. A session is a specific execution of an experiment, controlled by one specific experimenter, involving one specific subject, under an unspecified number of experimental conditions (this number can even be zero), with a specific beginning and end in time, labelled with a major and minor sequence number. A session is usually associated with a data file that contains user track data. This is one typical session:

seq.	subseq.	Start	end	experimenter	experiment	subject	conditions	datafile
1	3	2007-06-20 23:18:31	2007-06-20 23:20:05	Enrico Fermi	Philadelphia	M. Marini	Panoramic Cylindrical	Track-1-3.txt

The only delicate point is that the relationship between an experimental session and the conditions is many-to-many: an experiment can be done under any number of experimental conditions, and the same experimental condition can occur in any number of experiments. This is a classic problem for relational databases controlled by SQL, as there is no way to directly specify a many-to-many relationship in SQL. The classical solution, implemented by SQLObjects, causes the creation of an intermediate table, also known as an xref table, that contains couples of sessions and conditions.

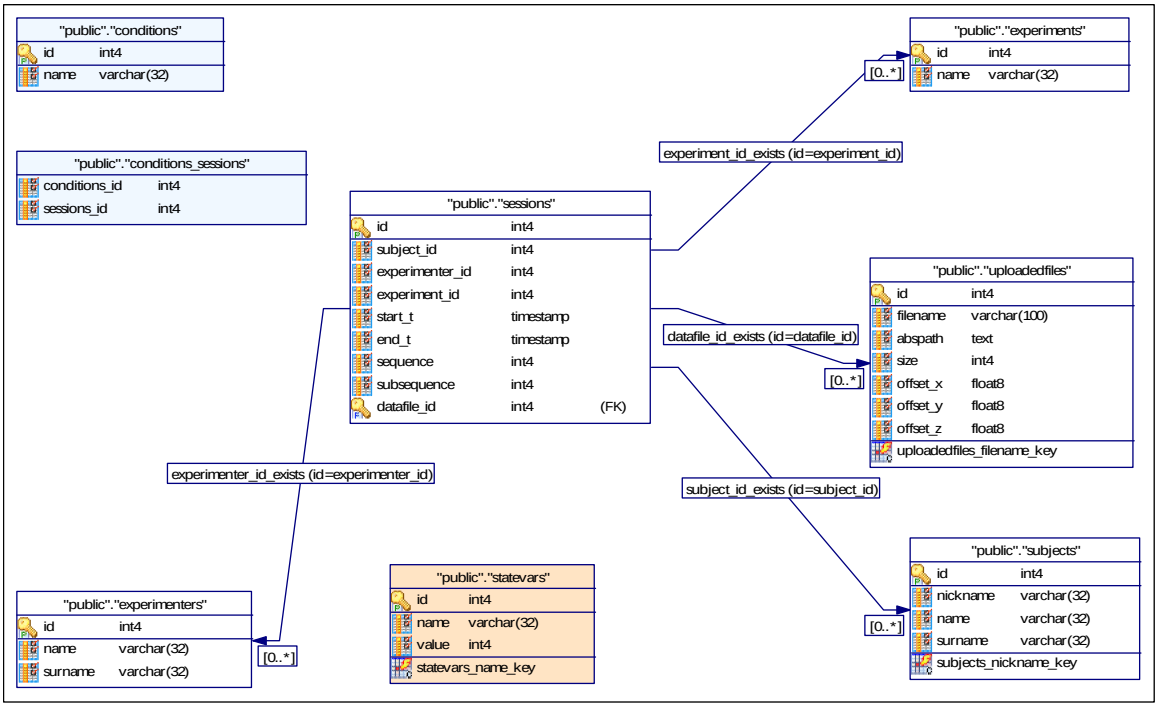

Figure 6: ER diagram

You can also notice a pink "statevars" table that allows for persistence of data between invocations of the software. This is convenient for storing information that is outside of the experimental model but needed by the software, like counters for generating filenames.

Trail data format

We have decided on a unified data format for storing user trail data. This format is a tab-separated file containing the following columns:

dt time difference from start of experiment
x coordinate
y coordinate
z coordinate
dx direction on X axis
dy direction on Y axis
dz direction on Z axis
s an arbitrary status string (for a user command or extra state information)

As the experimental applications have been written and used before Trailmaps was conceived, we have written a trailFilter class that will read raw logfiles from an open file-like object and write to a file-like object in the unified format that we have just described. Python's flexible duck typing lets us use exactly the same code when reading files and when reading a file uploaded from a webform.
It would be easy to write a trailFilter for data such as NMEA files.

Trail data files are referenced by the uploadedFiles class (and database table). This class also contains offset information that is used to shift data before aggregation and graphical representation. Offsetting data is necessary as a consequence of real life situations where the data may not share the same geometric origin. Data is stored and represented in 3D form, although currently the visit map algorithm works on a 2D matrix.

User interaction and performance

The user, currently not authenticated, uses the web interface to load experimental sessions and query the database to produce graphical reports and aggregated statistics. The application can be easily augmented with more refined forms of statistics.
Performance is currently quite slow, the majority of time being spent inside gnuplot. We will be looking into more efficient ways of representing visit maps, probably going from straight Python lists of lists to NumPy arrays, and into a way to replace gnuplot.

What is TurboGears?

TurboGears defines itself in the FAQ as '*… a rapid development "front-to_back" web meta-framework. Its aim is to simplify and speed up the development of modern web applications written in the Python programming language.*' It is generally compared to frameworks like Django and Pylons in the Python world, and to Ruby on Rails outside of it. TurboGears can be usefully contrasted with Zope, because it operates in the same problem area but uses a very different approach: while Zope is a very large and complex end-to-end develepment effort, going all the way from the database backend to the templating language, TurboGears is glue code holding together a collection of what the developers consider "best of breed" components: the Kid template language, the MochiKit collection of JavaScript libraries, SQLObject as an ORM (Object-Relational Mapper), and CherryPy as a dispatcher and a configuration manager. This is not a stable collection: in fact it is likely that in TurboGears 2.0 SQLObject will be replaced by SQLAlchemy. TurboGears (or, in fact, SQLObject) is quite database agnostic and it will interoperate with various SQL databases.

From the point of view of a Python programmer, TurboGears offers immediate satisfaction out of the box: once the programmer has understood the MVC (Model View Controller) pattern that underlies TurboGears, the responsibilities of the various parts of the application become clear, and development takes off. Thanks to MochiKit, there is no need to actually touch any JavaScript if you don't feel like it, and the SQLObject ORM layer makes database tables behave more or less like classes protecting your eyes from SQL – at the same time, basic proficiency in databases is very useful for understanding the possibilities and constrains of SQLObject classes.

Iterative development and future work

Thanks to having a few current and perspective users of the software in our laboratory, we can test often and early, and incorporate user suggestions in the software at an early phase. Since we were new to the TurboGears framework (although not to Python nor to databases) we felt that diving right into development would be more productive than producing a detailed spec based on a faulty understanding of the problem and of the capabilities of the platform.
The next features we want to add to the software are stored "views" of the data, where a view is a map with a fixed bounding box and a canned SQL SELECT statement, more ways to manipulate the data in order to allow more complex comparisons, and an off-line importer for loading large numbers of experiments into the system in a non interactive manner.

We are not distributing the system yet, but we plan to place it under GPL as soon as the documentation is at a level that would allow other people to work on it. This should happen in Fall 2007.

Bibliography

1. Avizzano, C., Gutierrez, T., Casado, S., Gallagher, B., Magennis, M., Wood, J., Gladstone, K.,Graupp, H., Munoz, J.A., Arias, E.F.C., Slevin F., *Grab: Computer graphics access for blind people through a haptic and audio virtual environment* in Proceedings of Eurohaptics 2003, 2003
2. Bresenham, J. E., *Algorithm for Computer Control of a Digital Plotter*, IBM Systems Journal, 4(1):25-30, 1965. Reprinted in Seminal Graphics: Pioneering Efforts That Shaped The Field, Rosalee Wolfe ed., ACM SIGGRAPH.
3. Chittaro L., Ieronutti L., *A Visual Tool for Tracing Behaviors of Users in Virtual Environments* in Proceedings of AVI 2004: 7th International Conference on Advanced Visual Interfaces, ACM Press, New York, May 2004, pp.40-47
4. Ieronutti L., Ranon R., Chittaro L., *High-Level Visualization of Users' Navigation in Virtual Environments*, Proceedings of Interact 2005: 10th IFIP International Conference on Human-Computer Interaction, Springer Verlag, Berlin, September 2005, pp. 873-885.

5. NMEA 0183 Interface Standard, version 3.01, National Marine Electronics Association, 1994, revised 2002.
6. Portillo Rodriguez, O., Avizzano, C. A., Bergamasco, M., Robles De La Torre, G., *Haptic rendering of sharp objects using lateral forces*, in proceedings of the 15th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN06), Hatfield, UK, September 6-8, 2006
7. Wu, X. *An efficient antialiasing technique*. Computer Graphics 25 (4): 143–152., 1991