JyVis - A Flexible High Level Visual-Analytic Framework

Curran Kelleher, Georges Grinstein

Abstract—The Universal Visualization Platform (UVP), developed at UMass Lowell, is a general Java-based framework for information visualization and analysis. The UVP contains many advanced and novel features added by various researchers over the last six years, and many visual-analytic tools of the UVP have been tailored for use in specific applications. Tool extentions and modifications were difficult to implement due to the various requirements placed on them by the system, the most burdensome of which being selection interaction and session monitoring support. We developed a new framework, JyVis, to support ease of tool modification, extention, and development, even for non-developers. JyVis is a visualization framework based on an architecture comprised of two layers: a low-level visualization framework written in Java, and high-level program elements written in Jython (a freely available implementation of the high-level, dynamic, object-oriented language Python seamlessly integrated with the Java platform). The key features that distinguish JyVis from other visualization systems are that all JyVis tools are themselves Jython scripts, pluggable, and end-user-programmable. JyVis provides support for session logging and replay. This is accomplished by having all significant state changes occur via scripts, and recording those scripts to a cumulative session history file for later replay. The ability to define tools with a small amount of code is enabled by a rich set of high-level visualization primitives (graphical objects which automatically support selection interactions) and UI widgets (UI building blocks which automatically support session logging and replay). JyVis architecturally supports most features of the UVP, including multiple visualizations, multiple selections, multiple datasets, linking and brushing of selections between visualizations, pluggable visual-analytic tools, session logging with the ability to replay previous sessions, kinetic displays, sonification, and voice annotation. The ability to access and use tools across the internet, and the ability to have a shared system state across many distributed users via simple text communication is also architecturally supported by JyVis.

Index Terms—Visual-analytics, Visualization, Script-based, Architecture.

1 Introduction

It is challenging to design a visualization framework that meets the needs of all of its potential users. Perhaps the biggest challenge is to provide the user with the ability to easily customize or change a visualization and its interactions so that they fit with the domain in which the user is most interested. The designer of the framework cannot anticipate all the needs of users, so the framework itself must be flexible enough to allow users to modify and create visualizations to suit their needs. In order to guide the design process of JyVis, we present a list of requirements that we feel a visualization framework must meet. We first list the requirements, then we explain how the JyVis framework meets these design goals.

1. Classic Visualization Requirements
   - Multiple Visualizations
   - Brushing and Linking of selections across visualizations
2. Modern Visual-Analytic Requirements
   - Multiple Datasets and Multiple Selections
   - Tools which interact with datasets and selections
3. Very Modern Architecture Requirements
   - Support remote tools
   - Easily modifiable visualization functionality
   - Easily create custom visualizations
   - Easily incorporate analysis tools
   - Script and log all actions for later replay

JyVis meets all of these requirements. The most fundamental aspect of JyVis is that it is split into two layers: a visualization framework written in Java, and high-level program elements written in Jython. Jython is an open source project which provides an embeddable pure-Java Python interpreter, and allows Python code executed by it to access Java libraries. Python scripts cannot access Java libraries the way Jython scripts can, so the language will be referred to throughout this paper as Jython. All high-level code that is executed in JyVis is written in Jython, and typically makes service of the visualization framework written in Java. All Jython code is executed through a singleton embedded interpreter, which itself is part of the Java framework.

There is a global "base environment" which contains a base window and a list of selectable datasets. Multiple visualization windows exist as internal frames within the base window. The currently selected dataset determines which data is used when a visualization or analysis tool is instantiated. For multiple selections, each dataset contains a list of selections, each selection itself being list of records. When a selection changes, the dataset itself sends each visualization a notification signaling it to redraw the selected records appropriately.

All "tools" are implemented as Jython scripts. Scripts are detected by the environment as "plugins" at run time, making all tools "pluggable." This allows users to modify existing tools and create new ones, and have them available to run without compilation.

The ability to access, create, and modify both datasets and selections is exposed in the API. A simple Jython script can take selections, and creates datasets out of them. The API exposes the ability to access and manipulate both datasets and selections, so scripts can be written which perform arbitrarily complex operations on datasets and selections.

To accomplish session logging and replay, the following restriction must be placed on all UI elements: every UI event is actually a script that gets passed through the Jython bottleneck, which executes the script in the embedded interpreter, and then stores the script to a session history file. Since this history file consists of only the previously executed scripts, it can be read in at a later time and executed one script at a time, so as to effectively "replay" the recorded session.

The next few sections describe the motivation for JyVis and more details.

2 Background

2.1 The UVP

The Universal Visualization Platform (UVP) [3], a general framework for information visualization and analysis, was developed by

• *Curran Kelleher and Georges Grinstein are with UMass Lowell IVPR, E-mails: curran_kelleher@student.uml.edu, grinstein@cs.uml.edu.*

students in our research group at University of Massachusetts, Lowell. Over a period of about six years, many advanced and novel features were added to the UVP by various researchers. It was used as an experimental visual analytics test-bed, and was applied to a wide variety of application areas. For these reasons, the UVP has become academia visual-analytics test-bed. The UVP features include multiple visualizations, multiple selections, linking and brushing of selections, pluggable visual-analytic tools, session logging [4], kinetic displays [8, 10], sonification [9], voice annotation [4], user modeling [4], and a recommendation system [2].

Even though the architecture of UVP does support pluggable tools, it is extremely difficult to develop these tools for two main reasons: all of the low-level graphics and interaction (selection for example) must be implemented entirely by the plugin at the lowest level, and the framework requirements that the visualization are forced to implement are overly complex. For example, a visualization tool must maintain its own "UVPModel" and "session data," which are necessary for the session framework to work. Also, any visualization tool must subclass "VisualizationTool," from which methods which return the property and legend panels must be implemented, thus requiring a large understanding of the UVP class structure. Finally, the plugins must be written in Java and compiled along with the entire framework to modify them or create new ones. This makes modifying tools, making new tools, and experimenting with tool ideas extremely difficult when compared to using JyVis.

In terms of lines of code, the definition of a scatterplot and its UI in the UVP takes a few thousand lines of Java code, where the equivalent tool in JyVis takes a few hundred lines of Jython code. We can accomplish this in our high-level Jython tool layer by leveraging the low-level Java visualization layer.

2.2 Processing

Fry and Reas developed Processing [7] to simplify the coding on interactive graphics. Processing is a programming language and environment which provides a rich high-level API for working with interactive graphics, animation, and sound. Processing targets artists and designers who want to create high-level interactive graphics programs and are not very familiar with programming. A notion central to Processing is that powerful programs can be written very few lines of code, so that end-users are able to understand it and use it effectively.

2.3 Plug-and-play Visualizations

North and Shneiderman [5] developed Snap-Together Visualization to make it easy to create custom visualization environments that include multiple coordinated visualizations. The way this is carried out is based on the relational database model. A key feature of Snap-Together is that new visualizations that are created can be easily integrated into the envoronment using just a few "hooks."

2.4 Summon

Rasmussen at MIT's CSAIL built a visualization prototyping program using Python, C++, and OpenGL called Summon [6]. The C++/OpenGL part of the system exposesd high-level graphic primitives, such as polygons, lines, and text labels, as well as a global drawing window to which those primitives can be added. The program features a Python console for entering script commands and executing scripts from files, rather than a GUI. This console interfaces to a Python interpreter embedded in the C++ program. The end result is the ability to write Python scripts which generate graphic primitives and add them to the drawing window to create visualizations. A Python function can also be associated with a graphic primitive, which is called when the user clicks on that graphic.

The user interaction inside the drawing window includes scaling and stretching of the visualization on the screen, as well as callbacks to Python functions when the user clicks on a graphic. Summon was designed to support visualizations of phylogenetic trees, sequence alignments across species, and other bioinformatics visualizations. There are two key features of Summon we considered: that it employs Python to create visualizations, and that this is done by using high-level graphic primitives which are incorporated into a low-level graphics system implementation.

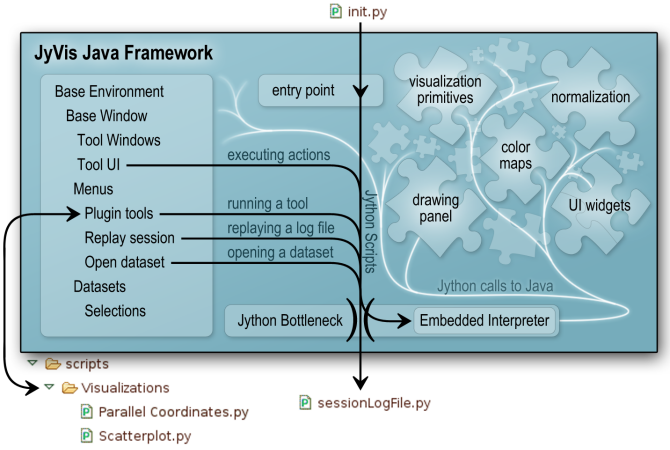


Fig. 1. The overall JyVis architecture

3 Architecture

A key feature of a visualization framework is that it is flexible enough to meet task-specific needs. The UVP is extremely powerful, but with that power is also complexity which hinders the tailoring of tools to task-specific requirements, and development of new tools. This is a major setback to potential users of the UVP, and has been a limiting factor to its success.

Another key feature of a visualization framework is that it contains enough prepackaged high-level building blocks to be useful in the arena of visual analytics. Summon is intended to make it easy to create visualization prototypes, which it does well, but nothing more. Summon has no framework for data management, record selection, or UI widgets.

We draw on the strengths of the UVP, Summon, Processing, Snap-together, Jython, and Java to create a new platform for visual analytics which provides a solution to the aforementioned issues. We use Jython and Java as opposed to Python, C++, and OpenGL for cleaner code, and to leverage the utility of the Java runtime library.

In an attempt to build a modern visual-analytic platform which is as powerful as the UVP, and as elegant as Summon, we developed JyVis. Our architecture is comprised of two layers: the visualization framework written in Java, and the high-level program elements written in Jython. The Java portion exposes a high level visualization framework to the Jython scripts. This framework written in Java provides an API for dealing with graphics, data management, selections, and UI widgets (which tools can use to build their UI).

Visualization and analysis tools are all implemented as Jython scripts. These scripts are detected by the system at run time and incorporated into the system as plugins, which are able to be run from within the system.

The program entry point is in Java, and all scripts are executed by a singleton embedded Jython interpreter. We impose the restriction that all significant changes to the state of the system happen by way of executing a Jython script. All scripts are executed by what we will call the *Jython bottleneck*. When a script is passed through the Jython bottleneck, it is executed by the embedded interpreter, then written to a session history file. This way, a log of all significant state changes will be recorded, which can be replayed later one step at a time.

3.1 Java Framework and API

3.1.1 Visualization Primitives

High level visualization primitives are the building blocks for all visualizations. Some visualization primitives include circles, lines, rectangles, polygons, and text labels. They are called visualization primitives

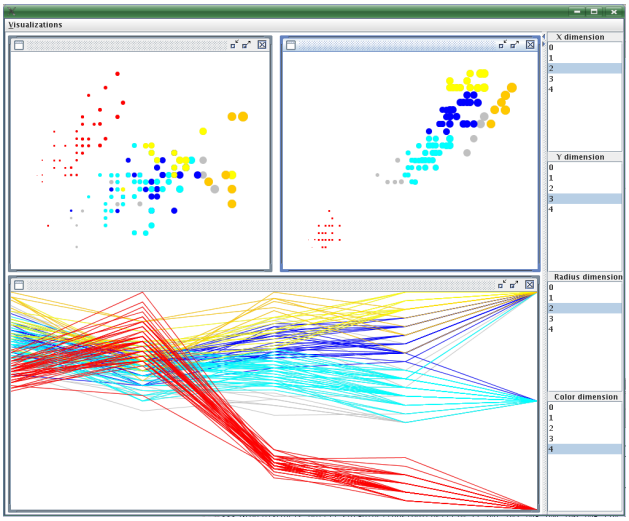


Fig. 2. A screenshot of the JyVis framework in action, visualizing the Iris dataset [1], running on Linux with KDE. Colors denote five selections which have been made, gray items are non-selected.

rather than graphics primitives because they support selection, and are designed specifically for use in creating visualizations.

There is a drawing panel onto which visualization primitives may be added. The default coordinate system used by the visualization primitives has both x and y range from 0 to 1. This window can be modified by the visualizations if desired. The transformation from coordinate space to screen space when drawing the visualization primitives to the screen is done automatically.

3.1.2 Data Sets

Data sets are represented internally as a list of data records, where a data record is a list of data entries. A data entry always has a numeric value, and optionally has an associated string. When building a data set from a file, each non-numeric data entry is assigned an ID number which is unique within its dimension. This data set structure allows for variable-length records and allows both tabular as well as semi-structured data to be used in a uniform fashion. If the data is tabular, the maximum and minimum of each dimension (dimension metadata) is calculated and stored when the table is created.

3.1.3 Normalization

In order to represent data entries visually, a normalization is often necessary. This is a function which maps the data values to a specified range, which for us is the canonical [0, 1], but can be any range. The normalization function often depends only on the minimum and maximum of the data entries to be mapped, but sometimes other information about the data is required (for example, global normalization requires the bounds of the entire dataset). A general normalization framework is provided, which specifies a method, `normalize`, which takes as arguments a record index, a dimension index, and a reference to a data set. This way, the implementations of `normalize` will have access to the entire data table and its dimension metadata when doing the normalization calculation.

3.2 Color Mapping

A color map is defined by a list of color-value pairs. A color map, once defined, provides a function taking a value between 0 and 1 as input, and outputs a color. The color which is output is calculated by linearly interpolating between the colors of the two color-value pairs whose values are closest to the input value.

3.2.1 Selections

Selection is implemented by maintaining a list of selections inside each data set. Each selection is a list of selected records. Each data record object contains a lookup table where keys are drawing panels, and values are lists of the visualization primitives. The visualization primitives are drawn by and selectable via their corresponding drawing panel in this lookup table. A list of visualization primitives is used, as opposed to just a single visualization primitive, so that visualizations are not restricted to using a single graphical object to represent one record. For example, this design allows a parallel coordinates implementation to represent one record by many lines (which are visualization primitives). To know which graphics to represent as being selected, the drawing panel iterates through the list of selected records in its associated data table, and queries each record to get references to the actual visualization primitives that it should draw as being selected.

When the selection polygon changes, or when the final selection is made, the list of objects which are inside the selection polygon is determined. This is implemented as a brute force search through all record-representing visualization primitives, asking each visualization primitive to compute whether or not it will be selected by the selection polygon. This selection-testing is implemented as a simple inside-test, testing whether or not an (x,y) point, such as the center point of a circle, is inside the selection polygon (this function is provided by Java). Selection determination can be optimized in the future by using spatial indexing when performing selection queries (such as a quad tree).

To set up a new visualization primitive with the selection framework, three related entities must be linked together: the visualization primitive, the data record, and the drawing panel. Specifically, the visualization primitive must be added to the list of visualization primitives which is mapped by the drawing panel (which in turn will draw it and be able to select it) held in the data record object. A method which does precisely this is provided by the visualization primitive base class, which takes as arguments a data record and a drawing panel. When the list of selections held in a data table is modified, that data table sends a notification to all of the drawing panels which reference it such that they update the display to represent the current selections.

3.2.2 Base Environment UI

The class `JV` contains the base environment as well as high-level convenience methods. The base environment basically consists of a base window, a list of datasets, and the data set that currently has focus. Static (global) methods for accessing the base environment are provided by `JV`. The way tools get a window for themselves is through a method in `JV` which sets up an internal frame inside of the base window.

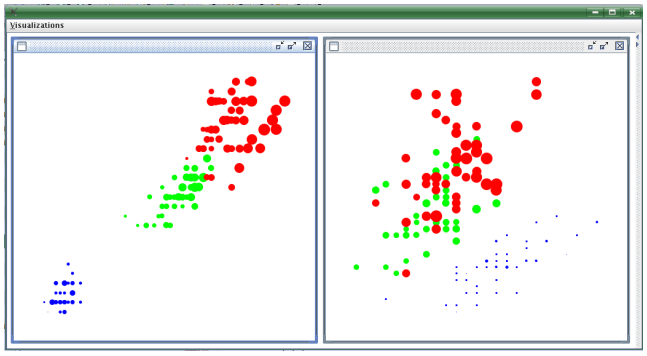
3.3 Plugin Architecture

To make JyVis as flexible as possible, all high-level code is written in Jython; this consists of the script that initializes the environment, and the tools that modify the environment (and may do other things too). The main entry point to our system is held in a Java class whose `main()` method does only one thing, executes the initialization script.

When the system starts up, a menu is created that mirrors the tree structure of the directory `/scripts/Visualizations`, whose leaf menu items, when clicked, will execute their corresponding script in the embedded interpreter. This menu is added to the base window. This is how tools are detected as "plugins" and incorporated into the base environment. This enables the end user to modify existing tools and write their own new tools, and use them without ever requiring compilation. Generating such a menu is a function provided by the API, so the initialization script can be modified to create other menus based on script directories, such as the *File* menu and others. It may be useful for visualization and analysis tools to leverage this function as well.

3.4 Session logging and Replay

A key feature of our architecture is session replay. To accomplish this goal, we impose the following restriction: that every significant state modification event that can occur takes place by way of executing a Jython script through the Jython bottleneck, which executes scripts in



```
# Generates a scatter plot. The arguments specify the
# indices of the dimensions used for calculating the
# x, y, radius, and color of the circles
def scatterPlot(xDim,yDim,rDim,cDim):
  #get the data
  data = JV.getSelectedData()
  if(data is None): return
  # create the visualization window
  frame = JV.createWindow(data)
  # the size of the margin and the min and max radii
  margin = 0.1; rMin = 0.002; rMax = 0.02
  # set up the linear normalization and color map
  normalization = LinearNormalization();
  colorMap = ColorMap.getDefaultColorMap()
  # plot every record as a circle
  for record in range(data.records.size()):
    # calculate the x, y, radius, and color values
    normalization.setOutputRange(margin, 1-margin)
    x = normalization.normalize(record,xDim,data)
    y = normalization.normalize(record,yDim,data)
    normalization.setOutputRange(rMin, rMax)
    r = normalization.normalize(record,rDim,data)
    normalization.setOutputRange(0, 1)
    c = normalization.normalize(record,cDim,data)
    # create a circle for he current data record
    o = JVCircle(x, y, r)
    # calculate the circle color with the color map
    o.color = colorMap.getColor(c)
    # set up the circle to work with selection
    o.setUpForSelection(data.records[record],frame)
    # add the circle to the window
    frame.add(o)
  # show the frame
  frame.setVisible(1)
# create the first scatterplot
scatterPlot(2,3,0,4)
# create the second scatterplot
scatterPlot(1,0,3,4)
```

Fig. 3. Scatterplot.py, a simple scatter plot implementation using the JyVis framework. The graphic above is the result of running this script.

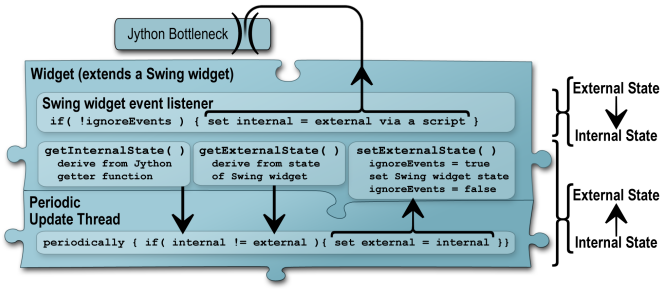


Fig. 4. The JyVis widget framework

the embedded interpreter and saves them to a cumulative session history file. This session history file, the log of all scripts which have been executed in the session, can be read in at a later time and iterated through, executing each script in the file so as to effectively replay the entire recorded session. The session history file is a text file, a valid Jython script in fact (.py), and all (potentially multi-line) scripts that are logged are delimited by a unique string, for example `"\n#!%JyVis-Script-Delimiter%!\n"`.

We must decide how to handle the case when a script triggers a UI event which causes the execution of another script, before the first script is finished executing. We now discuss an example of when this happens, which will lead to the solution. Consider the matter of replaying the action of opening a data set. This state-changing action happens (thus a script is executed) in the middle of execution of another script. It happens inside the scatter plot script in figure 3 when the line `data = JV.getSelectedData()` is executed. The behavior of the method `JV.getSelectedData()` is conditional on the existence of a focused dataset in the following way: if a dataset is loaded and has focus, then it is returned, otherwise, the user is prompted to open a dataset, the dataset file they choose is loaded into the environment, given focus, and returned (if no dataset is chosen, then `null` is returned).

Scripts executed by the interpreter are global level (global scope) scripts, meaning that they are never inside any nested code structure. Global level scripts with nested code structure (such as the definition of classes of functions) can only be executed in their entirety, and not one line at a time. Single line execution of nested code is not supported by the Jython interpreter. When the scatterplot script in Figure 3 is executed for the first time, it causes the Jython bottleneck to call itself recursively (when the user opens a dataset). During replay, this recursive call is "flattened" into sequential calls.

The question is, which script to log first; the inner one (the dataset opening script), or the outer one (the full scatterplot script)? Consider both cases when the log is being replayed. If the outside has been logged first (thus is executed first), then the full scatterplot script will be run before the dataset opening script, triggering the framework to prompt the user to open a dataset. This is the wrong behavior. If the inside is first, then the dataset will be opened first, and then used by the scatterplot script. This is the correct behavior. Thus when logging sessions, the inner script will always be logged before the outer script. This decision yields the execute-first, then log-second ordering in the Jython bottleneck.

3.5 UI Widgets

We have developed a general framework for session-enabled UI widgets. For a widget to be session-enabled means that it will do two things: generate and execute scripts when the user interacts with it, and change itself (its appearance) when those same scripts are executed later (when a session is being replayed). These widgets are the building blocks that tools will use to build their UI. Widgets in JyVis typically subclass widgets from the Java Swing library, such as check boxes, list selectors, combo boxes, sliders, radio buttons, text fields, and others.

The widget framework can be thought of as being based on the

```
# define a class to store the internal state, "a"
class testClass:
  def __init__(self):
    self.a = 1
  # define a function to set the internal state
  def setA(self,a):
    self.a = a
# create an instance of the test class
test = testClass()
# make and resize a frame to put the checkboxes in
frame = JFrame(bounds = (500,500,80,80));
# add the setter function to the GlobalObjects list
setterIndex = GlobalObjects.add(test.setA)
# create a getter function
getter = lambda: test.a
# create the checkbox widget
check = JLCheckBox(getter, setterIndex,"A")
# add the checkbox widget to the frame
frame.getContentPane().add(check)
# show the frame
frame.setVisible(1)
```

Fig. 5. A simple example script which uses the JyVis check box widget, called JLCheckbox (which stands for Jython Linked Checkbox).

model-view-controller paradigm (1). Each widget has an "internal state," the value of the model, and an "external state," the value of the view/controller (what is shown on the screen by the actual Java widget). The internal state is accessed via setter and getter Jython functions, which the widget is given access to upon construction. The widget uses the getter function to determine it's internal state, and the setter to set its internal state to match it to the external state when the user interacts with the widget. Swing widgets have their own model, view, and controller. These are different than our "internal" and "external" states. The entire Swing widget (including its model) in our context is considered to be the view/controller.

When the user interacts with the widget, it programmatically generates and executes a script which sets the internal state to be the same as the newly changed external state. When that script is executed later, when a session is being replayed, we have a situation where the internal state is not the same as the external state, and the external state should be updated to reflect the internal state. This updating of the external to reflect the internal is done in an indirect manner. There is a thread which periodically (every second) tests all widgets for consistency between their internal and external states. If the internal and external are different, out of synch, then the external state is updated to reflect the internal state (as this is happening, events may be generated that would otherwise generate scripts, so the script-generating mechanism must be temporarily turned off, typically by setting an `ignoreEvents` flag). To connect the widgets to the update thread, all widgets register themselves with the update thread upon construction.

There is one loose end. The scripts that set the internal state (to match the external state) must be executed at the global level, through the Jython bottleneck. However, the setter functions are often are members of objects, thus not accessible from the global level. So how do you generate a global level script which will call a non-global setter function? Somehow the setter function must be made accessible globally. This is done by adding the setter function to a global list, and using the index of the function in that list to access it with a global script.

4 Future Work

Currently, JyVis does not directly support kinetic displays, sonification, or voice annotation. However, the overall architecture does not require any changes to add these extensions. We plan to add support for kinetic displays [8, 10], sonification, text annotation, voice annotation, and the ability to export visualizations as vector graphics by the time of this publication.

When brought into the context of the web, the plugin and session architectures enable some interesting things. Since the tools are fairly small, they can be imported as plugins from a URL. This means that distributing a newly developed tool would require only posting it on the web, and having the end user enter its URL inside of JyVis, at which point the script is downloaded and executed, up and running immediately. The session architecture enables two systems to be synchronized via bidirectional text communication over the internet. When the user on one system takes an action, the event-script generated by that event is broadcast to the other user's system, which interprets and executes it immediately. The end result is effectively one system state which is shared between two users. This concept can be extended to include an arbitrary number of users, using something like a chat room. If all data sets and tools are accessed via the internet, then the resulting session history file is universally replayable, any system anywhere can replay it.

The JyVis framework is general enough to be a powerful tool for generating high-level applications outside the domain of visual analytics. In this context, perhaps the most relevant feature of JyVis that it allows developers to create session-enabled user interfaces with very little effort. JyVis may prove useful in building front ends, session-enabled or not, for existing libraries, which could be themselves written in Java or any language accessable by Java, including C++, JavaScript, Python, Ruby, Scheme, Groovy, Jaskell ("Java Haskell"), and many others.

JyVis is being made open source and will be available this summer from http://www.cs.uml.edu/~grinstei/JyVis.

Acknowledgements

The authors wish to thank Henry Kostowski, Damon Berry, Howie Goodell, Adam Fraser, Fanhai Yang, Matt Rasmussen, and Manolis Kellis.

References

[1] E. Anderson. The irises of the gaspe peninsula. *Bulletin of the American Iris Society*, 59:2–5, June 1935.

[2] H. G. Chih-Hung Chiang and G. Grinstein. Supporting visual exploration: beyond just interaction. submitted to the VAST 2007 Conference, 2007.

[3] M. Y. M. S. U. C. H. G. V. G. C. L. J. Z. C.-H. C. Gee A., L. Li and G. Grinstein. Universal visualization platform. Jan. 2005.

[4] C. K. A. B. Goodell H., C-H. Chiang and G. Grinstein. Collecting and harnessing rich session histories. In *International Conference on Information Visualization*, pages 117–123, July 2006.

[5] C. North and B. Shneiderman. Snap-together visualization: a user interface for coordinating visualizations via relational schemata. In *AVI '00: Proceedings of the working conference on Advanced visual interfaces*, pages 128–135, New York, NY, USA, 2000. ACM Press.

[6] M. Rasmussen. Summon - visualization prototyping and scripting. http://people.csail.mit.edu/rasmus/summon/index.shtml, 2006.

[7] C. Reas and B. Fry. Processing: a learning environment for creating interactive web graphics. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Web Graphics*, pages 1–1, New York, NY, USA, 2003. ACM Press.

[8] G. G. Smith, S. and R. Pickett. Global geometric, sound, and color controls for iconographic displays of scientific data. In *SPIE/SPSE - Symposium on Electronic Imaging - Extracting Meaning from Complex Data: Processing, Display, Interaction*, 1991.

[9] R. D. B. Smith, S. and G. Grinstein. Stereophonic and surface sound generation for exploratory data analysis. In *CHI'90 Empowering People - Conference on Human Factors in Computing Systems. Also in M. Blattner and R. Dannenberg, Eds. Multimedia and Multimodal Interface Design. New York: ACM Press 1992*, pages 125–132, July 1990.

[10] R. P. R. B. A. B. A. G. Yang F, H. Goodell and G. Grinstein. Data exploration combining kinetic and static visualization displays. In *International Conference on Coordinated Multiple Views in Exploratory Visualization*, July 2006.