visualeyes

Project XML Overview

Bill Ferster

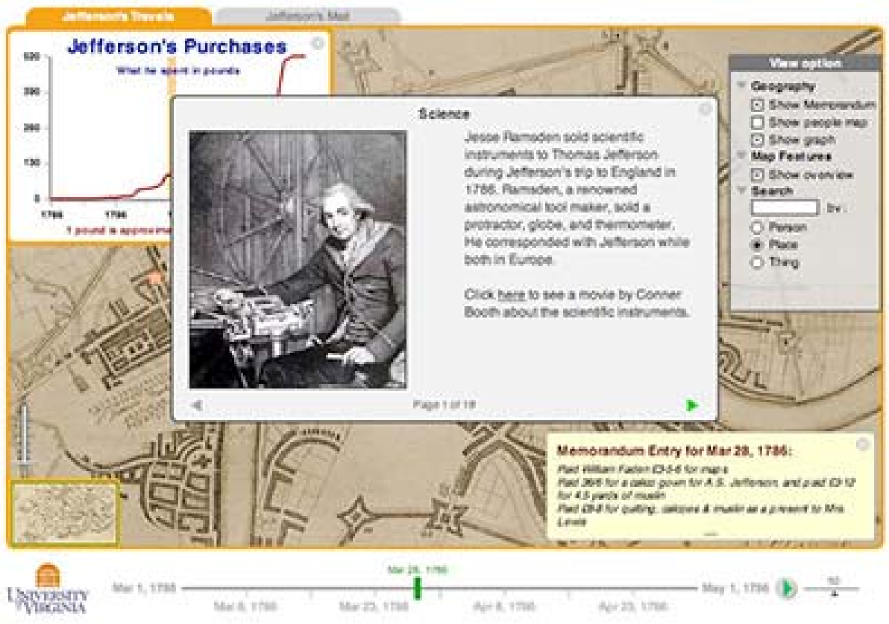
2/15/2010

VisualEyes allows users to interactively browse spatial and temporal events in a number of ways. It is a *generative browser*, allowing users to not only view preset collections of events, but to construct their own views of the events based on selected criteria. VisualEyes makes it easy to construct complex queries about events, weaving maps, timelines, and data visualizations to encourage insight.

VisualEyes is a tabbed-based collection of views of events and data that can be interactively shown within a time period using the timeline tool. Views can contain event descriptions, primary-source documents and imagery, maps, digital movies and audio, animations, charts and graphs of historical data.

Each view represents the result of choices of what to show in that view. The views can show events that match certain criteria, ranging from “show all events in Antioch, Virginia” to a sophisticated query such as “show all events where Jefferson bought more than 3 trees from 1796 to 1820, but not ones from Thomas Mayne.”

These views can be fixed for demonstration purposes, or left open, for people to explore various relations between the elements provided allowing for both purposeful and serendipitous discovery of complex interrelations.



Science

Jesse Ramsden sold scientific instruments to Thomas Jefferson during Jefferson's trip to England in 1786. Ramsden, a renowned astronomical tool maker, sold a protractor, globe, and thermometer. He corresponded with Jefferson while both in Europe.

Click here to see a movie by Conner Booth about the scientific instruments.

Memorandum Entry for Mar 28, 1786:

Paid William Faden £3-5-6 for maps
Paid 36/6 for a calico gown for A.S. Jefferson, and paid £3-12 for 4.5 yards of muslin
Paid £8-8 for quilting, calicoes & muslin as a present to Mrs. Lewis

Resources Supported

There are three basic kinds of information (resources) VisualEyes can display and search for:

1. **Maps -** VisualEyes contains a fully interactive geographic information viewer to display three basic kinds of maps. 1) Scans of historical maps, 2) vector-based maps from GIS systems such as arcGIS, and 3) maps delivered from Internet-based web services, such as Google Maps, or any combination of the three. All maps can be easily panned and zoomed, with an option to see an overview inset. Shape data can come from specified tables in a SQL/mySQL database, a link to an XML file, or from Internet-based web service.

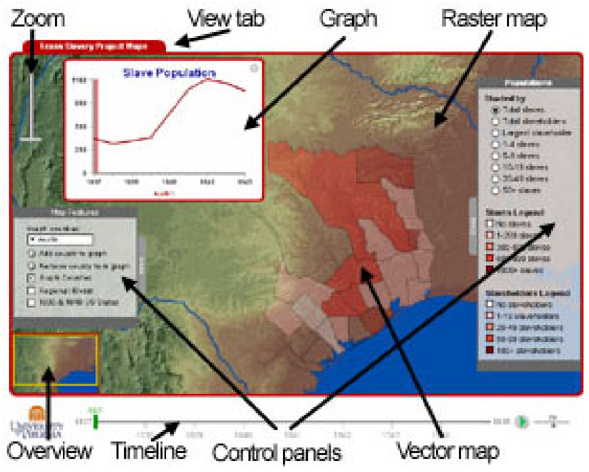
2. **Data -** A rich array of historical data can be imported into VisualEyes from a database as a table. This data can be displayed as a layer on a map, shown as a chart, table, or graphic element, and most importantly, used to include or omit events in the views. Data can come from specified tables in a SQL/mySQL database, a link to an XML file, or from Internet-based web service.

3. **Images/Movies -** Digitized images of primary source documents from digital archives can be displayed and integrated into maps, animations and other visualizations. These images can be JPEG, GIF or PNG formats, and can be dynamically sized and positioned, movies on YouTube; and Flash movies and animations.

View Components

An unlimited number of views can be constructed from these three basic resource elements. Views can be interactive, enabling users to change how the various resources interact, or static, allowing for a didactic interpretation of the events.

The infrastructure for a view contains a procedural description of how the information is displayed with conditional comparisons and loops so that very sophisticated queries can be performed. These queries are easily constructed using pull down menus for the various options desired indicating the relationships between the various resources.



1. **Control Panels -** Each View has its own pull-out area docked to a side of the screen that can be expanded or collapsed as needed and contains a number of collapsible check boxes to toggle on and off various features of the map, such as data overlays, roads, town names, etc. Various map features, such as the overview navigation insert and map legend can be turned on and off here as well, but assigning some "Glue" to be activated on clicking. Control panels can contain:

   - Radio Buttons
   - Check Boxes
   - Combo Boxes
   - Slider Controls
   - Text Input Boxes
   - Buttons
   - Headers and Legends

2. **Timelines -** Each view can have its own timeline that can control the temporal aspect of the project. Sliding the cursor changes the view's date, which in turn can change the way in which information is displayed if it is time dependent.

3. **Animation Players -** The current time on the timeline can be animated over time, using a player control, allowing the project to animate any time dependent elements from any point on the timeline to another.

4. **Zoomers and Overviews** – The screen can be controlled by a zoom slider and/or a small overview inset that facilitates panning through the screen.

5. **Graphs** – Various types of graphs (line, bar pie, scatter, etc.) can be drawn using dynamically generated data base on data from XML or SQL databases, time from timeline, settings of control panel items, or any combination of them.

6. **Tables and Text Displays** – Dialog-box based tables and text displays can be drawn using dynamically generated data base on data from XML or SQL databases, time from timeline, settings of control panel items, or any combination of them.

7. **Paths** – A series of positions on the screen (specified by pixels or latitude, longitude if a map) can be defined to appear at particular times. Each position can be marked by an icon, shape or image file. Clicking on the position can call up a web page draw graphical elements, or popup window showing some information. Lines can be drawn to connect these positions.

8. **Radial Maps** – A path can be arrange in a radial concept map format to help visualize relationships between objects shown.

VisualEyes XML Guide

VisualEyes is an empty vessel for interacting with historical information. Its entire functionality and "look and feel" is controlled by an XML data structure. This flexibility will allow it to be effectively used in a wide variety of projects, while still maintaining a common internal structure.

The topmost level on the hierarchy is the project. The project contains any number of views. This will allow for multiple interactive representations to be shown simultaneously, making for easy comparisons, with a shared timeline for control.

Each browser may contain any number of views. These views are represented as tabbed areas on the screen. Clicking on any of the tabs will bring up a different view. Each view contains descriptors of the *resource* to display, or use as data to change the display. There is one timeline for each view, which makes it easy to set the temporal aspect of that view.

- **Resource**
  The resource could be a *map*, some *media*, a table of *data*, or a *graphic*. Each resource item contains a query instruction as how to find the data for that resource. For example, a map may be a URL to a bitmap, a SQL query for a collection of shape files in a database, or a URL to an online web service, such as Google maps. A data table from a SQL database, from an XML file or a web service can be similarly used.

  A number of objects, such as timelines, charts, tables, graphs, etc. can be displayed on the screen using resources and glue defined below.

- **Glue**
  Glue (The General Language to Unite Events) is a procedural description of how the various resource elements connect with one another and are displayed. VisualEyes knows how to render a number of types of resource, such as tables, charts, text area, movies, audio clips, vector and raster maps, and the Glue language contains elements to cause them to display. The **project** and **controlpanel** rely on Glue to know how to display the views and sub-views.

  Glue also contains elements for linking user-generated actions, such as clicking on the screen with actions. Glue also provides an opportunity to calculate tables and fields in resources based on a simple script in the tag. Many common types of operation can be defined between these elements, so that VisualEyes is able to relate rich data relationships between them and visualize them on a special and temporal basis.

- **Controlpanel**
  Each view contains a control panel that can be populated with a number of interface items, such as checkboxes, radio buttons sliders and header elements. This panel can be docked anywhere in the view or be free-floating. It can be always present, or opened and closed like a drawer
  .
- **Containers**
  Containers are collections of objects used to put information on the screen on top of a resource, such as an image or a map. There are currently three types of a container: A Path creates trails and navigation objects, a CMap makes concept maps, and a Canvas defines a scrollable area to contain images.

Projects

The <**project>** defines the top-most mode of a VisualEyes project. The <**frame** > item defines the boundaries of the view panels and the <**tab>** area combined. The <**textformat** > will define the default text formating used for all views if not re-defined.

The <**logo>** item defines a bitmap to use across all views, and is typically below the area defined by the <**frame>**. The <**tab>** defines the height, width, and colors of the tabs, if any. The **<view>** tags provide the content for all of the views within the project. There is no limit to the number other than what will fit horizontally as tabs.

```
<project title="name" host="url_of_host" >                      // Topmost level
        <textformat> textformat </textformat>                   // default text attributes
        <frame> frame </frame>                                  // frame of project
        <logo top="pixels" left="pixels" source="url" />        // logo
        <tab                                                    // defines tabs
          onCol="0xrgb"  offCol="0xrgb"                         // color of on/off tabs
          onTextCol="0xrgb"  offTextCol="0xrgb"                 // text color tabs
          hgt="pixels" wid="pixels"                             // size of tabs
          curView="number" />                                   // starting tab to see
        <view > view1-n* </view> …                              // tabbed view(s)
</project>
```

Views

Each tab in the project contains a <**view>**. The <**view>** contains elements that are displayed on the view's **screen**. The <**textformat>** will use the project's text formatting as it's basis and elements can be over-ridden. **<resource>**'s such as maps, images and data are loaded for display. <**controlpanel>** s are set up to provide user interface controls. The scope of any <**view>** is itself, meaning each <**view>** is an island unto itself.

Almost all of the other elements that make up a project are within one or more view elements and provide the top-level navigation control fro your project.

Views can also be invisible and not associated with any particular tab. By setting the *visible* attribute to "true" and giving it an *id*, you can use **<glue>** to cause a view to show within the currently active tab's screen space. See the **setview() <glue>** item and the section on the cookbook section for more information on this very powerful option.

By default, if an image on the screen is larger than the height, you can drag it up and down with the mouse when fully zoomed out. To inhibit this panning unless you are zoomed in, set the *pan* attribute to "false."

***NOTES ABOUT FORMATTING**

<items> are shown in angle brackets
attribute of an **<item>** are in italics
"values" of *attributes* of an item are in quotes

```
<view id="id" title="name" >                                 // Tabbed views of data  *
            <textformat> textformat </textformat>                   // overrided text attributes
            <resource> resource </resource> …                       // resources(s) for this view
            <controlpanel> controlpanel </controlpanel> …           // control panel(s) for view
            <timeline> timeline </timeline>                         // timeline for this view
            <glue> glue </glue>…                                    // connection mapping
            <path> path </path> …                                   // path(s) for this view
            <cmap> conceptMap </cmap> …                             // concept maps for this view
            <zoomControl                                            // zoom control for view
                  top="pixels"   left="pixels"                      // position (omit to dock it)
                  def="number"                                      // starting value
                  max="number"   />                                 // max zoom (% /100)
            <overview                                               // overview navigation control
                  docking="[ botLeft | topLeft | botRight | botLeft ]" // docking
                  wid="pixels"   boxCol="0xrgb:0xffff00" />         // width and control box color
                  def="[ true | false ]"                            // show at startup?
                   src="inset.jpg" />                               // inset map image
            pan="[ true | false ]"                                  // allow panning of screen
            visible="[ true | false ]"                              // visible or free-floating view
</view>
```

***NOTES ABOUT FORMAT OF XML ITEMS**

Italics indicate link to another XML object
… indicates there may be multiples of these objects
underlined option indicates default

Control Panels

Controlpanels provide a dialogbox-like means for setting parameters of the screen. These parameters can be set using **items** such as check boxes, radio buttons, combo selection boxes, sliders, text input, and buttons to cause some sort of action. Items typically cause some action by adding an id of a **glue** tag to call when they are changed or clicked.

The ***frame*** specifies the size, position and color of the **controlpanel**. The *docking* attribute in the ***frame*** can be *top, left, bottom, right, or float*. The first 4 cause the **controlpanel** to "stick" to that side of the screen, ignoring the positioning parameters (top & left) set in the **frame**. The setting the *docking* to "float" will allow the ***controlpanel*** to be positioned anywhere on the screen as dictated by the "top" and "left" attributes.

The **textformat** will use the **view**'s text formatting as it's basis and elements can be over-ridden. The title sets the name in the **controlpanel** 's bar. Setting the *closable* attribute adds a handle to collapse the panel into the screen's frame. The *open* attribute sets the default status at startup of the **controlpanel** either open or closed.

```
<controlpanel title="name" >                                     // Control panel for views
        closable="[ true | false ]"                              // enable panel closing tab
        open="[ true | false ]"                                  // panel closing startup status
        <frame> frame </frame>                                   // box of control panel
        <textformat> textformat </textformat>                    // default text attributes
        <item id="name" > …                                      // each line in the panel
                type="[ checkbox | radio | slider | textbox |    // item type (buton is not a typo!)
                        header l legend | buton | buttonbar | line | text | combobox | half ]"
                def="value"                                      // default value/state of item
                title="name"                                     // name of item to show
                linkto=""                                        // itemID that can make it visible
                bold="[ true | false ]"  italic="[ true | false ]"  // is text bold or italic
                <glue> glue </glue> *                            // glue object to run if clicked
                min="value"   max="value"                        // min/max values (slider only)
        </item>
</controlpanel>
```

*built-in glue objects "*legend*" & "*inset*" toggle on/off if in a "*checkbox*"

The **items** form the active portion of the **controlpanel** as a running list of objects. They are drawn using the current font settings, but the *bold* and *italic* can be over-ridden on a per-item basis. The current *leading* in the **textformat** item determines how far they are spaced vertically. If there are more items than fit vertically, a new column is started.

Radio items act a a collective group, allowing only one radio on at a time. Header items will collapse items below it (until the next header item). Legends provide a box colored by the def attribute, and are stacked from the bottom of the ***controlpanel*** up. Setting the glue of a checkbox item to "legend" will cause the legends to collapse if checked

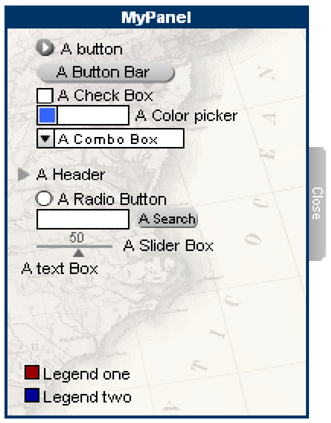
You can change control panel items dynamically by using the **menuitem()** GLUE method. You can also see what a given item's value is set to by using the item's id. For example, `status(*myCheckBox)` would print "1" if checked, and "0", depending on the checked state of a **controlpanel item** named "myCheckBox."

The *linkto* attribute is used to control the visibility of an item based on the setting of another item, such as a checkbox or radio button. This is useful in control which items are visible in the control panel when certain options are checked on or off, say hiding the controls for a chart when the radio button showing the chart is unchecked.

Types of Control Panel Items

buton	A round button that will trigger a glue method when clicked (not a typo!)
buttonbar	A square button with the title written inside that will trigger a glue
checkbox	A checkbox with the title to the right that will trigger a glue method when clicked.
color	A color chip to choose a color from a set of choices, or type the RGB values
combobox	A combo box to choose between several choices
half	Used to add a half-space vertically (leading) to the list
header	An arrow control to collapse or expand the items the follow until another header
legend	Used to put a color choice when drawing legends
line	Draws a separator line
query	Adds a query line (if: something equals value)
radio	A radio button, of which only one is active in a contiguous group of them
search	A text input box with a search button bar attached
slider	A horizontal slider to set the value from 0-100
text	Displays a line of text
textbox	A text input box

Here is an example of a control panel with many of the items added:



```
<controlpanel title="MyPanel">
      <textformat leading="16" />
      <frame docking="left" alpha="75" wid="200" hgt="300" />
      <item title="A button" type="buton" />
      <item type="buttonbar" title="A Button Bar" />
      <item title="A Check Box" type="checkbox glue="myGlue" />
      <item title="A Color picker" type="color" />
      <item title=""Yes|No|Maybe" type="combobox" />
      <item title="A Half Line" type="half" />
      <item title="A Header" type="header" />
      <item type="legend" def="0x990000" title="Legend one" />
      <item type="legend" def="0x000099" title="Legend two" />
      <item title="A Radio Button" type="radio" />
      <item title="A Search Box" type="search" />
      <item title="A Slider Box" type="slider" />
      <item title="A text Box" type="text" />
<controlpanel>
```

Working with ComboBoxes

Combo boxes are useful items that contain a series of choices in a drop-down menu. When an option is selected, its name is set as the current value, just like a 0 or 1 is set with a checkbox. You set the values in the title attribute, separated by | marks like this:

```
<item title=""Yes|No|Maybe" type="combobox" id="myCombo" glue="myGlue"/>
```

The glue can then get the currently selected option like this:

```
<glue id="myGlue">
    status(*myCombo)
</glue>
```

Timeline

The **timeline** tag will add a graphical timeline that will allow the user to set a time period along a horizontal timeline using a slider bar. A play button can be added to the timeline to animate the setting of the slider bar over time by setting *play* to “true.” Setting *speed* to “true” will put up a speed control aside the play button, with a default speed of 50%. That default can be over-ridden by setting the speed to some other number from 1-100.

The **frame** tag set’s the size and position of the timeline on the screen. The *wid* attribute sets the width, and *backCol* sets its color. The **textformat** will use the **view**’s text formatting as its basis and elements can be over-ridden.

The *min*, *max* and *start* attributes define the minimum , maximum and starting dates for the timeline, and can be expressed as “year”, “month/year” or “day/month/year.” Dates can be formatted as years (1976), month/year (3/1856), day/month/year (3/7/1756), or full date (January 6, 1798).

The timespan can be divided by 4 major tick marks, whose length is set by *majorTick* and 3 minor tick marks set by *minorTick*. The tics can be place above, below and across the main timeline bar by setting *tickPos*. The *showValues* and *showMinorValues* determine if the tick mark’s time values are displayed.

Labels signifying specific dates can be added using a labels tag. What direction they emanate from the main bar is set by *pos*, and the distance away from the main bar is set by *offset*. Lines can connect the label to the main bar by setting the *lines* attribute. .The ***textformat*** will use the ***timeline’s*** text formatting as its basis and elements can be over-ridden The actual entries are set using the ***timelinelabels*** Glue method (see GLUE section).

The *speed* attribute controls how fast an animation will play. The default is 100, meaning that it will play 10 seconds with the speed slider in the middle. Setting it lower, say to 20, will make the timeline play in 50 seconds, and 50 will play 20 seconds, etc.

```
<timeline>                                                // Timeline controller
        <frame> frame </frame>                                    // box of timeline
        <textformat> textformat </textformat>                     // specific text attributes
        <timebar>                                                 // for punctuated timeline
        play=”[ true | false ]”                                   // show play button
        speed=”[ Number:50 | false ]”                             // show play speed controller
        min=”date”    max=”date”                                  // start and end dates
        start=”date”                                              // starting slider dates
        dateFormat=”[ yr | mo/yr | dy/mo/yr | mo/dy/yr | mo,dy,yr]” // date format
        majorTick=”pixels:0”                                      // major tick make length
        minorTick=”pixels:0”                                      // minor tick make length
        tickPos=”[ top | mid | bot ]”                             // position of ticks rel to main bar
        numrTick=”Number:4”                                       // number of ticks on timeline i
        showValues=”[ true | false ]”                             // show values of major ticks
        showMinorValues=”[ true | false ]”                        // show values of minor ticks
        sliderDatePos=”[ top | bot | none ]”                      // slider date’s position
        <labels>                                                  // labels under timeline
                pos=”[ top | bot ]”                               // position of labels to main bar
                lines=”[ true | false ]”                          // show lines from bar to labels
                offset=”number:8“                                 // distance of labels from bar
                <textformat> textformat </textformat>             // specific text attributes
        </labels>
</timeline>
```

Punctuated Time Bar

You can add a segmented bar that will control the span of the overall timeline (called a punctuated timebar) by adding a **timebar** to the timeline. The timebar consists of a series of segments (must be contiguous!) that contain a starting and ending date within the overall *min* and *max* set in the timeline, and provides a way to make the timeline's span to be a portion of the full time being represented for better control and slowing animation of short events within the larger timeframe.

When a segment is clicked, the timeline's *min* and *max* settings are set to the segment's *start* and *end* attributes, making the timeline's span smaller. The screen is updated as if you had dragged the slider to the segment's start date. A button to make the timeline extend from the first segment to the last can be added by setting the *all* tag to true.

A glue tag can be specified to cause some glue to trigger when any segment is clicked. You can append parameters to the glue's name, which will be available as variables in the glue element.

For example, if we wanted to pass the start and end time to the glue, we would spec the glue as: **`glue="newTime?2&8",`** the value of 2 would be set in **`$$click`** variable and the value of 8 would appear in the **`$$param`** variable within the **`newTime`** glue element.

```
<timebar>                                          // Punctuated timeline controller bar
        <segment>                                          // contiguous segments
        glue="glueID"                                      // glue if clicked
        all="[ true | false ]"                             // show  "show all" button
        onTextCol="0xrgb"  offTextCol="0xrgb"              // color of active/inactive text
        onCol="0xrgb"  offCol="0xrgb"                      // color active/inactive segment
</timebar>


<segment>                                          // Timebar segment
        start="date"                                       // segment starting date
        end="date"                                         // segment ending date
        title="string"                                     // label of segment
</segment>
```

Resources

Resources contain information to be used by VisualEyes. This information is most often a table of data, but can be an interactive vector map, text, images, animation, movies, audio, charts, and graphs. ***Resources*** are the raw material for VisualEyes views. The ***<resource>*** tag in the project file provides a way to identify sources and provide *named access* to the data they contain. This access is useful because once they have been identified; we can refer to them by name later on using lines of Glue to easily create complex visualizations.

The various types of resources fall into four categories: 1) Image and Movie Resources, where images, movies and audio are loaded; 2) Data Resources, where numberic and/or string data organized into tables is loaded; 3) Map Resouces load vector maps from ***shapeData*** imported from arcGIS and other GIS and graphics packages; and finally 4) Graph and Table Resources that add charts, graphs, tables and other pop-up graphics to display data and information.

Common Resource Attribute Tags

All ***Resources*** have some common tag attribute elements, such as the *id*, *preload*, *type*, *title*, and *onclick* tags. The *id* provide a way to uniquely identify the resource to other elements in a ***view***. The scope of any resource is within the ***view*** it is contained by. If the *preload* tag is set, a spinning cursor will appear while that resource is being loaded. Thiis is useful when the display is dependent on an element, such a basemap to load. The type defines the type od resource, i.e. map, image, data, etc. The *onlick* attribute defines a ***glue*** element to execute when right-mouse clicked by the user.

```
<resource id=”name” >                           // Resource object
        title=”Name”                            // short title of resource
        type=”resourceType”                     // resource type
        src=”url”                               // web address of resource
        preload=”[ true | false ]”              // hold up start until it’s loaded
        onclick=”glue”                          // glue to run if clicked on
</resource>
```

1. Image and Movie Resources

Image Resources allow you to add JPEG, GIF and PNG images from any valid URL provided in the *src* tag. These images are added directly to the view’s screen (top-left corner by default, but can be anywhere, as set by *top* and *left* tags), where they can be panned and zoomed. Any number of images can be layer. Setting the *depth* to “topMost” will draw the image independent of any panning or zooming. Setting wid to non-zero, sets that image’s width to that size.

```
<resource>                                          // Resource object
        type=”[ image ]”                            // media type
        top=”pixels:0”          left=”pixels:0”     // spacial origin of image
        wid=”pixels:0”                              // width if non-zero
        depth=”[ screen | topMost ]”                // stuck to screen / freestanding
        frameCol= ”0xrgb”                           // color of image frame, if any
        frameWid=”pixels:0”                         // width of frame, if any
</resource>
```

Perhaps the simplest kind of resource is an image file, which might contain an image to display. The following line identifies an image on the server, loads it for future use, and makes it available to be instantly shown by referring to it by the name “myPic.”

```
<resource id=”myPic” type=”image” src=”http://virginia.edu/pic.jpg”/>
```

Movie / Audio / Animation Resources are Flash video formatted files (.FLV), MP3 audio files and SWF flash files. The *autoplay* tag which determines if the movie playing when it first appears. Omitting the *wid* tag will cause movie and player to size itself to match the native resolution on a Flash movie. Setting the *glue* to some glue object will cause that glue object to be called every *n* ms specified by *time*. *Start* and *end* specify the movies bounds.

```
<resource>                                          // Resource object
        type="[ movie | audio | flash ]"            // media type
        src="url"                                   // address of media file
        autoPlay="[ true | false ]"                 // play movie when visible
        autoRewind="[ true | false ]"               // rewind movie after end
        timer="ms:250"                              // time between glue calls, in ms
        glue="glueID"                               // glue to run each timer interval
        start="ms:0"                                // start of movie, in ms
        end="ms"                                    // end of movie MUST BE SET!
        <textformat> textformat </textformat>       // specific text attributes
        <frame> frame </frame>                      // box of timeline
</resource>
```

2. Data Resources

In its simplest form, a data resource is a list of things: numbers, words, paragraphs, URLs, etc. Data can be brought into projects in four ways. 1) By specifying the listing in the XML directly; 2) by accesing an XML file somewhere on the web via a URL; or 3) using a web service.

Each method stores the data into an identical format with VisualEyes, as a list containing the items, referenced from the resource's name. Data brought in this way can be queried using the GLUE query() method or accessed in a number of ways.

1. Defining data directly in the XML:

`<resource id="myData" type="sdata" src="1,2,3,4,5" />`
Will create a list of 5 items (1,2,3,4,5) within the resource accessible through myData.data1.

`<resource id="myData" type="xydata" src="10;5,20;6,30;7" />`
Will create a list of 3 items within the resource accessible through myData.data1 (10,20,30) and myData.data2 (5,6,7) useful for paired x y coordinate data.

2. Accessing data via an XML or CSV file:

`<resource id="myData" type="xml" src=http://mysite.org/data.xml/>`
Will load a file called data.xml on the server at mysite.org. That file can have any number of fields and rows. The project tool has a converter that takes tab-delineated spreadsheet files and formats it automatically. The actual format is listed in the appendix. The data is accessed by its field's name (i.e. myData.censusAge). You can load a CSV (Comma delimited) data file as well, but this will add 2-5 seconds to the load time when the project is run.

Accessing individual data elements in a table

Tables are typically accessed by querying the data with a query() method, but you can access individual elements by specifying them by field. For example**,** if we had a resource with the *id* of "myTable" and a field called "name", **status(*myTable.name)** would print a list of all the rows of the *name* field on the screen and **status(*myTable.name.1)** would print the 2nd name (the count starts at zero).

3. Map Resources

```
depth="[ screen | topMost ]"                    // stuck to screen / freestanding
onclick="glueID"                                // glue to run if feature clicked
ondoubleclick="glueID"                          // glue to run if double-clicked
onhover="glueID"                                // glue to run if hovered over
cols[]                                          // list of feature interior colors
edg[]                                           // list of feature edge colors

<resource id="name" >                                    // Resource object
    type="[ data | map | image | graph | table | movie | audio ]"// media type
    query="sql"                                          // actual SQL query
    src="url[/db]"                                       // web address of resource
    store="[ live | new | local ]" *                     // data storage options
    xy="x[0],y[0]; x[1],y[1];… x[v],y[n]"                // xy  data
    preload="[ true | false ]"                           // hold upstart until it's loaded
    onclick="glue"                                       // glue to run if clicked on
</resource>
```

4. InfoBox Resources

Information boxes are popup boxes used to display textual information on demand. They are typically called by clicking on path and graph elements. InfoBoxes can contain a variant of HTML formatting and can be populated using search and replace variable that can be set using a database. The appendix contains detailed information on the text formatting options available. See the example in the ***Cookbook*** of how to set up an infoBox.

```
<resource id="name" >                           // Resource object
    Text to display
    title="Name"                                         // short title of resource
    type="infobox"                                       // resource type
    close="[ true | false ]"                             // has closing button?
    selectableText="[ true | false ]"                    // has selectable text?
    scroller="[ true | false ]"                          // has scroll bar?
    border="pixels:24"                                   // indented border around text
    backImg="url[/db]"                                   // web address of image
    tail="[ line | none | solid ]"                       // tail to click point
    position="[ abs | north | south | east | west ]"     // position relative to click point
</resource>
```

The text can be have special tags($$1 through $$99 that can be replaced dynamically using the **replaceword()** method. Text can contain the standard text formatting macros (see appendix).

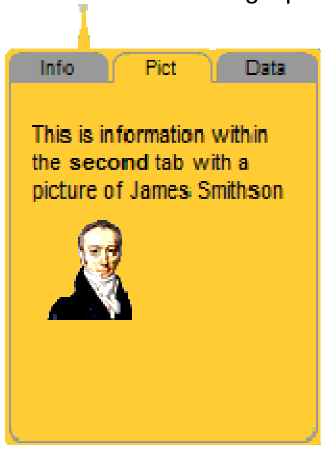
For example this script:

```
<resource id="myInfoBox" type="infobox" tail="line" >This is $$1 and this is $$2 />
<glue id="fillIt" from="myInfoBox" >
        list($v,one sentence,the second)
        replaceword(myInfoBox,$v)
</glue>
```

Would result in a box with the following text: "*This is one sentence and this is the second"*

Tabbed InfoBoxes

InfoBoxes can be subdivided into multiple tabbed pages, to make it easier to display larger amounts of information in a smaller space. Each section has its text divided by a header() tag. The name with the tag's parenthesis will show up in the tab. For example:



Info
Pict
Data
This is information within the second tab with a picture of James Smithson

```
<resource position="south" tail="solid" id="eventBox"
type="infobox">

        <frame backCol="0xFFCC33" corner="6" hgt="200"
wid="160" />

        header(Info)This is information within the b(first) tab

        header(Pict)This is information within the b(second)tab
with a  picture of James Smithson
        img(http://www.viseyes.org/smithson/sm-icon-tran.gif)

        header(Data)This is information within the b(third) tab

</resource>
```

5. DocViewer Resources

An **docviewer** is resource very similar an *infobox* to that can hold HTML formatted text and a picture side-by-side in series of pages provided by a data source (i.e. and XML file or SQL query). The data source can have 4 fields: *title, source, desc* and *caption*. The *title* field provides a title at the top and a way to select items from the data source. Items with the same title will appear as pages within the document viewer. The *source* field gives a url for a picture if desired, and *desc* is an html formatted text area. If a *caption* field is defined, it will appear underneath the picture.

If both *desc* and *source* are defined, they will appear side by side. If only one is defined, only that one will appear. The text and picture information is supplied by the *filldocviewer* glue method, typically as the result of a query method. Text can contain the standard HTML formatting macros (see appendix).

See the section in the appendix on “Making Booklets” for more information on creating the contents of docviewers/

<resource id=”name”**>**	**// docviewer object**
selectable=”[true | false]”	// text can be selected by mouse
scroller=”[true | false]”	// has a scroller bar
nopan=”[true | false]”	// inhibit panning on vertical pix?
numpos=”[top | bot]”	// where page numbers appear
type=”docviewer”	// set as docviewer
<textformat> *textformat* </textformat>	// overall text attributes
<frame> *frame* </frame>	// box of callout
</resource>	

For example this script:

```
<resource id="myData" type="xml" src="myDataFile.xml"/>
<resource id="myDocView" type="docviewer” />
<glue id="fillIt" from="myDocView " >
        filldocviewer(myDocView,Overview,myData)
</glue>
```

Would load an XML file called *myDataFile.xml* , when *fillIt* was called the docviewer would be filled by any pages in the xml file with the title of “Overview.”

Dots

Dots are an item used by a number of display items to put information on the screen on top of a resource, such as an image or a map. There are currently three types of displays that use <dot> items: A **Path** creates trails and navigation objects, a **CMap** makes concept maps, a **Dock** mimics the action of the Apple Macintosh dock. The basic idea of a **<dot>** item is that it describes 1) A graphical element that appears on the screen 2) in a particular place, 3) at a particular time and 4) call <g**lue>** methods when you click or hover over them with your mouse.

1. **Graphical Attributes**

    a. ***style*** sets how the dot's graphic will appear on the screen when shown. There are a number of built-in drawn types, such as:

        i. **bar** - A solid box that can be sized and colored
        ii. **cir** - A solid circle that can be sized and colored
        iii. **star** - A solid box star can be sized and colored
        iv. **rbar** - solid box with rounded corners that can be sized and colored
        v. **triu, trid, tril,** or **trir** - Solid triangles facing various directions
        vi. **icon:** - A vector image that can be sized. See the appendix for styles available

    You can also specify graphics images via a url. VisualEyes support images in the JPEG, GIF, SWF, and PNG formats. You must specify the full Internet address: (i.e. *style="http://www.mysite.com/mypic.jpg*").

    b. ***col*** sets what color built-in drawn styles such as *bar* and *cir* will be drawn in.
    c. ***alpha*** sets the transparency of the graphic from 0 (invisible) to 100 (fully opaque)
    d. ***lab*** sets a label to be written by the graphic (*labelPos* sets its position relative to graphic)
    e. ***frameCol, frameWid*** - Sets a colored border frame on graphics images.

2. **Spatial Attributes**

    a. ***x*** sets where on the screen the graphic will be drawn horizontally. When used in Concept Maps and Docks, the x and y attributes will be set automatically for you. In geo-rectified paths, the *x* sets the *longitude* value.
    b. ***y*** sets where on the screen the graphic will be drawn vertically. In geo-rectified paths, the *y* sets the *latitude* value. See "geo-rectifying maps" in the appendix for more info.
    c. ***wid*** is the width of the graphic, in pixels. If you do not specify a *hgt*, the height will be set in proportion to the width, so the aspect ratio is maintained.
    d. ***hgt*** is the height of the graphic, in pixels.
    e. ***rot*** sets the angle the graphic will be drawn at.
    f. ***labelPos*** - Set where, in relation to the dot graphic, the label will be drawn if anything is in the *lab* attribute. *labelPos* can be set to one of these values:

        i. **bot** - Label is drawn *centered* below the graphic
        ii. **top** - Label is drawn *centered* above the graphic
        iii. **left** - Label is drawn *flush-left* to the left the graphic
        iv. **right** - Label is drawn *flush-right* to the *right* the graphic
        v. **center** - Label is drawn *centered* inside the graphic

    Setting *labelPos* to anything else will cause the label not to be drawn.

3. Time Attributes

a. ***date*** - Sets the date when the <dot> will begin to appear on the screen. Can be specified in any of the following date formats: "yr", "mo/yr", or "dy/mo/yr".
b. ***end*** - Sets the date when the <dot> will begin to disappear on the screen. Can be specified in any of the following date formats: "yr", "mo/yr", or "dy/mo/yr".
c. ***pct*** - Used by **<pathway>** to specify when <dot>s will appear relative to the **<route>** timings. See section on pathways and routes for more information.

4. Interactive Attributes

a. ***glue*** - Sets the name of a **<glue>** item to run if the dot's graphic is clicked on.
b. ***hover*** - Sets the name of a **<glue>** item to run if the dot's graphic is hovered over.

<Dot> Item Format

```
<dot    id="name">                                      // Dot marker object
        col="0xrgb:0x000000"                            // color of dot
        wid="pixels:0"   hgt="pixels:0                  // width / height of dot
        rot="degrees:0"                                 // angle of rotation
        alpha="opacity:100"                             // 0-100 opacity
        style="[ _ | bar | but | cir | star | triu | trid | tril | trir | rbar | icon: | .jpg/.gif/.png ]"// shape
        x="pixels"       y="pixels"                     // location for dot in pixels
        date="day/mo/year" –or-  time="0-1:-1"          // time 0-1 (-1=no time) or date
        end="day/mo/year"                               // end date
        pct="Number 0-1"                                // used in route only
        lab="string"                                    // label for dot marker
        glue="glue"                                     // glue to activate if clicked
        hover="glue"                                    // glue to activate if hover'd over
        labelPos="[ top | bot | left | right | center | none ]"   // position of label
</dot>
```

NOTES:

1. Dots will continue using properties set in previous dots to reduce unnecessary repeating of attributes. For example, if you set the style to "triu" (up-facing triangle), all dots that follow would be rendered as "triu" until re-specified.

2. Clicking on a dot's graphic will cause a **<glue>** item to run if there is one specified, allowing you to trigger other actions and displays. You can find out which <dot> was clicked by looking at the $$param global list parameter, which will be set to the <dot>'s index in the path. The first dot will set $$param to 0, the second to 1, etc.

   Alternatively, you can set this value manually by appending it the <glue>'s name with a ? mark, such as glue="myGlue?show me". This will cause the words "show me" to be set in the $$param list and available for use in the <glue> script.

PATH

Paths place **dots** on the screen and can be connected by lines if desired. The width, color, and alpha can be specified. The position of the dots is set in pixels, relative to the base resource the path is atop, or in lat / lon coordinates based on the base resource.

If *showAllDots* is set *true* the dots are not time dependent. If *true*, dots can have times associated with them, so they will appear when the view's timeline date reaches a certain time. The time or date attribute of a dot tells when that dot will be drawn. 0 is at start, .5 is middle, 1 is end, etc. If a date is set, dot will draw when that date matches date on timeline. Time dependent paths can have the line advance between dots as the time changes by setting *tweenLines* to *true*. The current time of the line can be preceded by an icon by setting the *head* attributes. Paths are useful in showing a trail on a map, but are often used to put buttons, menus and other navigational elements on the screen. Set *showAllDots* to *true,* so they will always appear See the example in the ***Cookbook*** of how to set up a path or a menu.

```
<path  id="name" >                                          // Path object
       col="0xrgb: :0x000000"                                   // color of inter-dot lines
       wid="pixels:0"                                           // width of lines (0 = none)
       alpha="opacity:100"                                      // 0-100 opacity
       res="resourceID"                                         // Resource to pull GIS info
       showAllDots="[ true | false ]"                           // show all dots always
       tweenLines= "[ true | false ]"                           // animate line between dots
       headStyle=" [icon:name | .jpg/.gif/.swf ]""              // Leading line icon style
       headSize="pixels"   headCol="0xrgb"   headRot="0"        // Leading line icon wid/col/rot
       headEnd= "[ true | false ]"                              // Leave icon on when done
       glue="glue"                                              // glue if head is clicked
       <textformat> textformat </textformat>                    // specific text attributes
       <dot> dot </dot> …                                       // marker dots
       <pathway> pathway </pathway> …                           //  pathways containing dots
       <route> route </route> …                                 //  draw pathways
</path>
```

If you have a number of journeys along a set number of path ways, you can define a collection of dots as a **pathway**. The timing within the path is relative from 0 to 1 start to end, rather than a particular date for better flexibility and accomplished by setting the pct attribute in the dots contained in the pathway. That **pathway** can be drawn multiple occasions and different times by adding **route** elements to a **path** that define the *start* and *end* times a particular *pathway* will be drawn. See ***Cookbook*** for example.

```
<pathway  id="name" >                                       // Pathway object
       <dot> dot </dot> …                                       // marker dots
</pathway>

<route>                                                     // Route object
       pathway="ID of pathway"                                  // id of pathway to draw
       start="date"                                             // route start date
       end="date"                                               // route end date
       glue="glue"                                              // glue if head is clicked
       col="0xrgb"                                              // over-rides path col line color
</route>
```

Clicking on a head will cause a GLUE element to run if there is one specified, allowing you to trigger other actions and displays. You can find out which head was clicked by looking at the $$click global list parameter, which will be set to the dot's index in the path. The first head in the first path will set $$click to 1000, the second to 1001, etc. Alternatively, you can set this value manually by appending it the glue's name with a ? mark, such as glue=myGlue?show me. This will cause the words "show me" to be set in the $$param list and available for use in the glue script.

CONCEPT MAP

Concept maps are similar to paths, but the paths can be arranged in a radial manner similar to a hub and spoke shape. The dots are not time dependent, and lines (edges) must be specifically drawn by setting the relationships between the dots (nodes). Labels are automatically drawn if specified underneath the dot. The frame specifies the overall bounds of the concept map.

You can add a **legend** that identifies the type of lines by including a legend tag. Each tag adds an entry that shows a label associated with each **linestyle**. Setting backCol will draw a "wash" of color alpha'd over the background, to help highlight the concept map.

```
<cmap id="name" >                                  // Concept map object
        shape= "[ radial ]"
        <textformat> textformat </textformat>              // overall text attributes
        wid="pixels:0"                                     // width map
        hgt="pixels:0"                                     // height (same as wid if omitted)
        stagger="pixels:0"                                 // stagger length of "spokes"
        backCol=0xrgb                                      // background "wash"
        <dot> dot </dot> …                         s       // nodes
        <line> line </line> …                              // edges
        <lineType> lineType </lineType> …                  // edge types
        <legend> legend </legend> …                        // legend entries
</cmap>
```

The **lines** define the relationship between the dots and determine how they will be placed. Setting the *from* tag to "" will position the dot pointed by the *to* tag it at the center of the concept map. The **linestyle** object defines how the lines will be drawn, and the letter that will be drawn in the middle. See the example in the ***XML Cookbook*** of how to set up a radial concept map.

Dots are typically arranged automatically, but you can arbitrarily place a dot anywhere on the screen by setting the *x* and *y* **dot** attributes to a position and setting the **line**'s *dir* attribute to float. If you have specified a line style, the line will be drawn from the center of dot specified in the **line**'s *from* attribute to the center of the dot.

```
<line>                                             // Connector line (edge) object
        from="dotID"                                       // node connected from
        to="dotID"                                         // node connected to
        style="lineStyleID"                                // type of connection
        dir="[ one | two | float ]"                        // direction
</line>

<legend>                                           // Legend
        style="lineID"                                     // id of line style
        lab="String"                                       // text to display
</legend>

<lineStyle>                                        // Connector lineType object
        col="0xrgb"                                        // color of dot maker
        wid="pixels:0"                                     // width of dot marker
        alpha="opacity:100"                                // 0-100 opacity
        letter="letter"                                    // width of dot marker
        lab="label"                                        // rollover text
        type="[ isa | partof | contains ]"                 // type of connection  (TBD)
        arrows="[ from | to | both | none ]"               // line ends (TBD)
</lineStyle>
```

DOCK

A **dock** object presents a series of dots horizontally across the screen in a similar fashion to the application dock used in the Apple Macintosh OSX. The dots are typically icons or images that are fixed to a base bar. As the mouse hovers over one, it and its neighbors grow by the percentage spec'd by the *growth* tag. Setting the *growStyle* to *"single"* will cause only the dot being hovered on to grow while hovered over, as opposed to the default of *"taper",* which also grows the two dots on either side of the one being hovered over as well. The dots can have glue attached to cause some action when clicked.

The *frame* object sets the bounds of the dock, but since the dock grows and shrinks based on the number of *dots* within it, the dock will draw from the center of area defined by the frame's *left* and *wid* tags. The frame's *hgt* tag defines the height of the base bar. Setting the *hgt* to 0 will inhibit the drawing of the base bar.

<dock id="name" **>**	**// Dock object**
<textformat> *textformat* </textformat>	// specific text attributes
<frame> *frame* </frame>	// box of base bar
growth="Percentage:200"	// size to expand when hovered
growStyle="[single **|** taper]"	// style of growth when hovered
<dot> *dot* </dot> …	// dots in container
</dock>	

TIMEVIEW

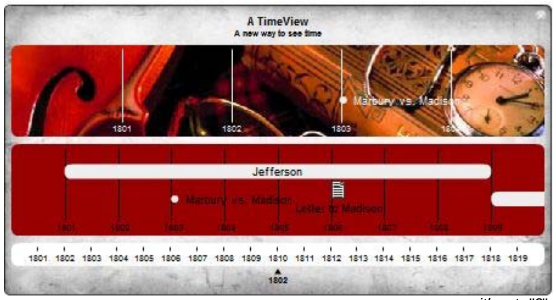
A TimeView is a display that shows events that are timed to occur at particular dates. It is similar to a traditional graphic timeline like MIT's Simile. A **timeview** item can have any number of **band**s, *each one having it's own time scale*, allowing you to show events that occur in vastly different time scales, such as decades, years and days. All the bands are linked, so scrolling one, scrolls the others.

Setting the *rot* attribute to something other than "0" will cause the timeline's bands to be wrapped around a cylinder in 3D. The cap of the cylinder can be a full oval or cut off at the top with the *capFull* attribute.

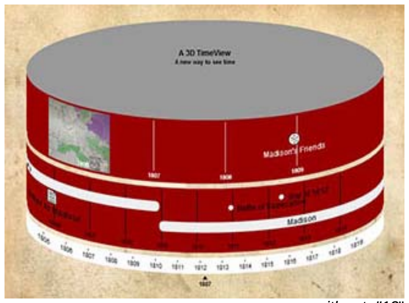
The bands are made up of individual events, each with a date, a label, an icon type, etc, just like dots are used in the **path** and **cmap** (concept map) displays, and fully clickable:

<timeview id="name"**>**	**TimeView object**
backImg=""	Background image URL for full frame
border="8"	Border amount in pixels
capCol="0x999999"	Color of 3D cap as an RBG hex number
close="true"	Has close button? options="false|true"
dateCol="0x000000"	Color of central date pointer as an RBG hex number
dateSize="0"	Size of central date pointer
drag="true"	Can drag timeview box" options="false|true"
fullCap="true"	Show complete oval as cap in 3D view?" options="false|true"
min=""	Starting time of the timeview in any time format
max=""	Ending time of the timeview in any time format
rot="0"	Angle of 3D rotation in degrees (0-45)
subtitle=""	Sub-title
timeline="true"	Sync to timeline in view?" options="false|true"
title=""	Title
<textformat> *textformat* </textformat>	
<frame> *frame* </frame>	
<band> band </band>	Band object(s)
</timeview>	

<band	id=”name”**>**	**// Band object**
	backImg=""	Background image URL for band
	border="8"	Border amount in pixels
	col="0xffffff "	Color of background as an RBG hex number
	corner="0"	Radius of corner of frame for making rounded rectangles
	hgt=""	Height of frame in pixels
	ratio=100""	Percentage of total time to show in band
	shading="50"	Shading amount in 3D (0-100)
	tickCol="0x000000"	olor of tick mark lines as an RBG hex number
	tickDateFormat="yr"	Format for tick options="yr|mo/yr|dy/mo/yr|mo/dy/yr|mo,dy,yr"
	tickDatePos="bot"	Position of tick line date text" options="top|bot"
	tickSpan="365"	Number of days between tick mark lines
	tickWid="0"	Width of tick mark lines in pixels
	<dot> *dot* </dot> …	Dots(s)
</band>		



with rot="0"



with rot="12"

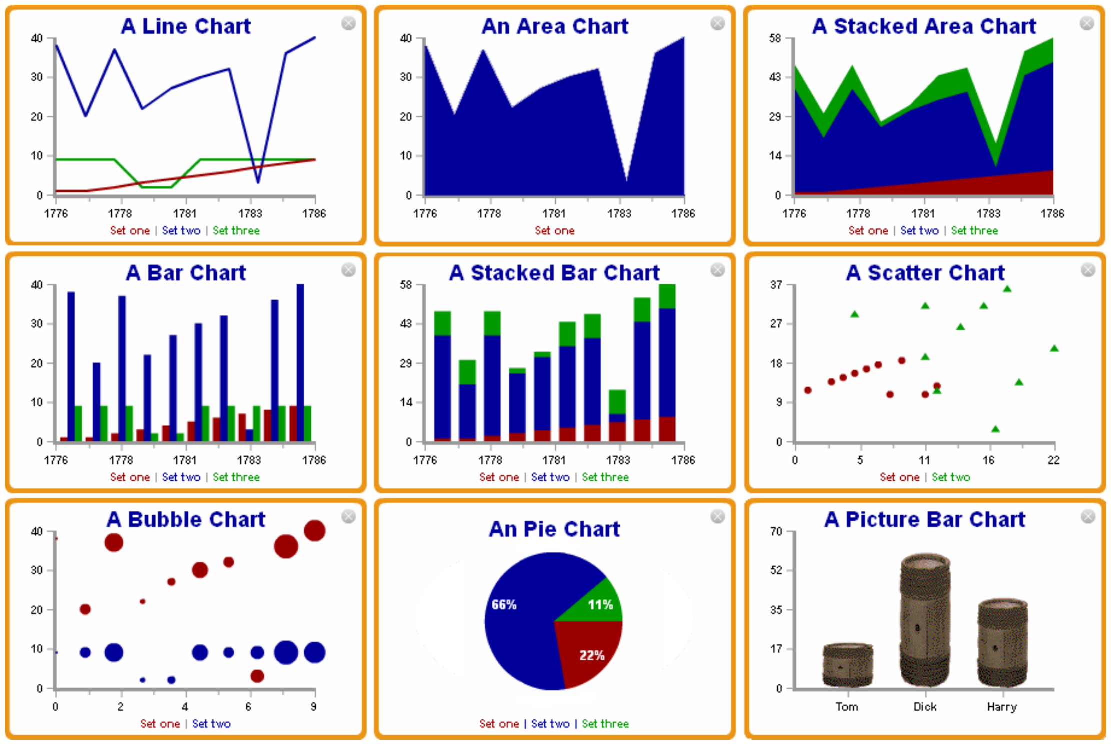
This example's script:

```
<resource dateSize="12" min="1800" max="1820" id="myTimeview" type="timeview" close="true" rot="0"
   backImg="greyparchment.jpg" title="A TimeView" subtitle="A new way to see time">
    <frame wid="600" frameWid="1" corner="8" left="100" top="100" backCol="0x9999999" />
    <band hgt="100" tickWid="1" tickCol="0xffffff" col="0x990000" corner="9" backImg="WatchPan.jpg" ratio="25"
        <dot style="cir" wid="8" date="1803" col="0xeeeeee" labelPos="right" lab="Marbury" y="60" />
    </band>
    <band hgt="100" tickWid="1" col="0x990000" corner="9" ratio="50" tickDatePos="bot">
        <dot style="but" wid="100" hgt="16" col="0xeeeeee" date="1801" end="1809" lab="Jefferson" y="30" />
        <dot style="but" wid="100" hgt="16" date="1809" end="1817" lab="Madison" y="60" />
        <dot style="but" wid="100" hgt="16" date="1817" end="1821" labelPos="center" lab="Monroe" y="30" />
        <dot style="cir" wid="8" date="1803 labelPos="right" lab="Marbury vs. Madison" y="60" />
        <dot style="icon:document" wid="15" date="1806" labelPos="bot" lab="Letter to Madison" y="50" />
        <dot style="cir" wid="8" date="1811" labelPos="right" lab="Battle of Tippecanoe" y="30" />
    </band>
    <band hgt="25" tickWid="1" col="0xffffff" corner="9" ratio="100" tickDatePos="bot" tickeCol="0x000099" />
</resource>

<glue init="true" from="myTimeview" />
```

Data Graphing

VisualEyes supports a number of chart types that can be drawn, including line, area, stacked area, bar, stacked bar, scatter, bubble, picture, and pie charts as shown here:



The charts can have multiple data sets, the color and labels of each are defined by the **marker** item. Charts can have x and y axis by adding an **xaxis** or **yaxis** item to the resource definition, The *title* and *subtitle* attributes allow you to add titles and subtitles to the graph. The bar and area charts can have their data sets stacked by setting the *stacked* attribute to true. Setting the *legend* attribute to true will show whatever **marker** *names* were set for that dataset at the bottom.

Scatter and bubble charts are bi-variate, requiring 2 datasets for each plotted set. On scatter charts, the first sets the position on the X-axis and the second one sets the position along the Y-axis. On bubble charts, the dots are plotted along the X-axis like a line chart, but the first data set controls the size of the dot drawn at each point. The bubbles are scaled according their value relative to the largest data value in that first set. The dataset's **marker** *wid* attribute sets the maximum size of the bubbles when the data value is the highest.

A line is automatically drawn between dots on a scatter chart. To turn off lines, set the marker's *datawid* attribute to "0" (the default value is "2").

Pie charts get their label names and colors from the **marker** tags. There should be one **marker** for each pie slice. You can have the slice values printing inside each slice by setting the *showValues* attribute to "true", or "percent" if you want the slice's percentage to the whole.

Use the **dataset()** GLUE method to set the data sets with values. You can easily animate charts by using the **tweenlist()** GLUE method to transition between two lists of data.

The charts pictured before were made using slight variations of the following script on the following page:

```
<resource id="myGraph" type="graph" style="area" stacked="false" border="35"
 title="A Chart" legend="true">
       <textformat col="0x000099" size="18" bold="true"/>
       <frame wid="300" hgt="200" left="24" top="160" corner="8" frameWid="4"/>
       <xaxis col="0x999999" majorTick="8" minorTick="6" wid="3"
        min="1776" max="1786" showValues="true" lab="one|two|three"/>
       <yaxis col="0x9999999" majorTick="8" wid="3" min="0" max="80" mod="1"
        showValues="true"/>
       <marker col="0x990000"/>
       <marker col="0x000099"/>
       <marker col="0x009900"/>
</resource>
<glue from="myGraph" init="true">
       list($myData1,1,1,2,3,4,5,6,7,8,9)
       list($myData2,38,20,37,22,27,30,32,3,36,40)
       list($myData3,9,9,9,2,2,9,9,9,9,9)
       dataset(myGraph,0,Set one,$myData1)
       dataset(myGraph,1,Set two,$myData2)
       dataset(myGraph,2,Set three,$myData3)
</glue>
```

<resource id="name"**>** **// Graph object**
close="[true | false]" // show close button
type="graph" // set as graph
glue="glueID" // glue if clicked
title="name" // title of graph
showValues="[true | percent | none]" // show data values in pie
subtitle="name" // subtitle of x axis
highWid="pixels" // width of highlight bar
type="[bar | line | area | scatter]" // type of graph
border="pixels" // space around data area
stacked="[true | false]" // are data elements stacked?
<legend show="[true | false]" /> // show legend
<textformat> *textformat* </textformat> // specific text attributes
<**marker** num="number" marker="*marker*" title="name" />… // data set info
<xaxis> axis </xaxis> // X axis
<yaxis> axis </yaxis> // Y axis
</resource>

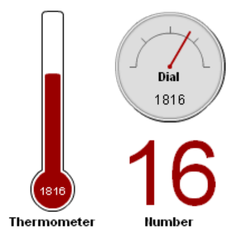
<marker id="name"**>** **// Marker object**
style="[bar | cir | trid | tril | trir | triu]" // type of marker
col="0xrgb" // color
wid="pixels:10 // width of marker
datawid="number:2" // width of data (i.e. line or bar)
labelPos="[top | bot | center | none]" // position of label
name="name" // name of marker
<textformat> *textformat* </textformat> // specific text attributes
</marker>

<xaxis> | <yaxis> **// Axis objects**
title="name" // axis title
col="0xrgb" wid="pixels" // color and line width
majorTick="pixels" minorTick="pixels" // major/minor tick lengths
grid="[true | false]" // show gridlines
valueCol="0xrgb" // color of values, if any
lab="a|b|c" // x labels separated by |'s

min="number" max="number" mod="number"	// data range and mod
showValues="[true | false]"	// show data values
valuePrefix="prefix"	// prefix to axis values
pos="[left | right]"	// poisition (yaxis only)
<textformat> *textformat* </textformat>	// specific text attributes
</xaxis>	

Widgets

Widgets are a type of graph that graphically displays a single continuous value on the screen, such as a dial, clock, thermometer, etc. The range of widgets available will grow with time, but they plot the *val* attribute from *min* to *max.*

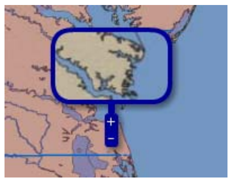


The data is plotted in the color *col*. The title is displayed below the widget except for the dial, where it's in the dial. The value is displayed to 2 decimal places if it is less than 1, or otherwise whole numbers. The size of round widgets like dials look at the *wid* attribute, where things like thermometer use the *hgt* attribute as well.

You can set the val in glue methods, like this `set(*myWidget.datVal,25),` which would set the value of the widget with the id of "myWidget" to 25. See the example in the cookbook.

<widget id="name"**>**	**// Widget object**
title="name"	// title
col="0xrgb:0x990000"	// color of inter-dot lines
style="clock|crop|dial|magnifier|number|thermometer"	// type of widget
wid="pixels" hgt="pixels" top="pixels" left="pixels"	// size and position
min="number:0" max="number:100"	// data range
val="number:50"	// Initial value
src="image source"	// Source for images
<textformat> *textformat* </textformat>	// specific text attributes
<frame> *frame* </frame>	// frame (for crop widget)
</widget>	

Magnifier Widget



The *magnifier* style widget will put magnified area of the screen atop a an image resource and let you drag it around as you would a real magnifying glass. Use the same image in the *src* attribute as the base image you want to magnify. See the "Appraisals" tab in the Vinegar Hill project to see one in action.

A **<frame>** item set the size and color of the frame and the *top* and *left* attributes set its initial screen position. Clicking on the "+" and "-" on the handle scale the zoom up and down.

```
<resource type="widget" style="magnifier" id="myMag" src="http://baseMap.jpg" >
     <frame wid="100" hgt="60" corner="20" top="100" frameCol="0x000099" frameWid="4" />
</resource>
glue init="true" from="myMag" />
```

Common Objects

```
<textformat id=”name”>                                  // Defines text formatting
        color=”0xrgb:0”                                         // text color
        alpha=”opacity:100”                                     // 0-100 opacity
        size=”pixels:12”                                        // size
        bold=”[ true | false ]”                                 // bold
        italics= [ true | false ]”                              // italics
        underline= [ true | false ]”                            // underline
        face=”[ _sans | _serif | _fixed | specificFont ]“               // font face
        slant=”degrees:0”                                       // orientation in degrees
        leading=”pixels:auto”                                   // total height between lines
        align=”[ left | right | center | justify ]”                     // horizontal alignment
</textformat>


<frame id=”name”>                                               // Holds visual items
        id=”name”                                                       // frame’s id
        title=”name”                                                    // title of frame
        wid=”pixels” hgt=”pixels” top=”pixels” left=”pixels”            // size and position
        corner=”pixels:0”                                               // for rounded rectangles
        alpha=”opacity:100”                                             // 0-100 opacity
        docking=”[ left | right | top | bottom | center | float ]”      // docking mode
        backCol=”0xrgb:0xffffff”                                        // interior color
        frameCol=”0xrgb:0x000000”                                       // color of frame
        frameWid=”pixels:0”                                             // width of frame
        dropWid=”pixels:0”                                              // width of drop shadow
        dropBlur=” pixels:0”                                            // blur of drop shadow (0-9)
        dropCol=”0xrgb:0x000000”                                        // color of drop shadow
</frame>


<shapedata>                                                     // Shape object
        col=”0xrgb”                                                     // default interior color
        edgeCol=”0xrgb”                                                 // default color of edge
        edgeWid=”pixels”                                                // default edge wid (0=none)
        xOff=”pixels:0”   xOff=”pixels:0”                               // offset of image to screen
        <polygon | polyline | arrow | text                              // element id
                id=”name”                                               // name of element
                xy=”xydata”                                             // cords (x,y; … x,y;)
                col=”0xrgb” edgeCol=”0xrgb” edgeWid=”pixels />          // color info
</shapedata>
```

Glue

In VisualEyes, using <glue> items is the heart of making interactive visualizations. This is the most difficult concept in VisualEyes to understand, but it is simple in principle. GLUE is an acronym, the General Language to Unite Events with two primary functions:

1. To cause resources, such as images, paths, and charts, to show up on the screen, automatically or on command.

2. To connect the data resources to data consumers, such as through display tables, popup windows, charts, and data-driven maps, using small scripts.

Screen Redraw

Because VisualEyes projects are highly interactive, the screen constantly needs to be redrawn to reflect the changing visualization. We call this a *screen redraw*, and it may be the result of clicking on a control panel item, scrolling of the timeline, or clicking on a screen it.

Your project is made up of a number of items such as such as a <resource>, <logo> or <frame> items within your project file. These items are loaded by VisualEyes when it first starts up and provide the "building blocks" your project will use.

Items such as the <controlPanel>, <timeline>, and <logo> show up automatically, but resources need to be "told" to draw themselves on a screen redraw, and that's what adding a <glue> item can do.

When a <glue> Item is "run"

A <glue> item is different from other items, in that it is active. <glue> items cause something to happen, such as an image to be displayed, some values retrieved from a data source, etc.

The screen is redrawn at startup, or as a result of a user's action, such as clicking on a control panel item or scrolling of the timeline. Each time the screen is redrawn, VisualEyes looks at the <glue> items in the view and if the <glue> is set to be activated, it will be *run.*

Being *run* means the <resource> the glue is connected (via the *from* attribute) to will be displayed, and/or the *script* within the <glue> item will be executed line by line. This occurs each time the screen is redrawn if the *init* attribute is set to "true." Glue can also be run by items such as checkboxes in a <controlPanel> by referring to its *id* attribute..

The Format of a <glue> item

A <glue> item is an item like any other item in VisualEyes, such as a <resource>, <logo> or <frame> item:

```
<glue  id=”name”
       from="name of resource"
       init=”false”
       once=”false”
       script (optional)
</glue>
```

There are four possible attributes to a glue item:

1. The *id* attribute allows you to give the glue item a unique name to call it by.
2. The *from* attribute specifies the resource to display on the screen
3. The *init* attribute causes the <glue> run each time the screen in drawn.
4. The *once* attribute cause the <glue> run only once (useful for initialization).

Aside from the four attributes, you can optionally add a *script* that will support calculating tables and fields within resources – and many common types of operations can be defined between these two elements within VisualEyes, to relate and display rich data relationships between them on a spatial and temporal basis.

You do not need to specify all of the attributes, as they have default values if left out. The *init* and *once* attributes are assumed false if not present, and unless the <glue> item will be called by an item such as a <controlPanel> checkbox, the *id* can blank.

A Simple Glue Example

The simplest case for using glue in VisualEyes is to get an image to appear on the screen. Assume we have created an image resource in VisualEyes called myPic:

```
<resource id="myPic" type="image" src="www.mysite.com/pic.jpg"/>
```

To make myPic appear, we need to "glue" it to the screen each time the screen is drawn, so we add the glue command below. It has the *init* attribute set to "true" and the *from* attribute set to "myPic":

```
<glue from="myPic" init="true" />
```

We did not need to name this glue with an *id* because it will be called each time the screen is redrawn. So when the user clicks on something, moves the timeline, or the project simply starts up, the image referred to by "myPic" will be drawn on the screen.

Glue Scripts

Scripts can be thought of as a kind of "to-do list" of things to be done in your project when the <glue> item is run, at startup or in response to some action your user has done, like a clicking on a control panel item, clicking on a map, or scrolling a timeline.

The lines on the <glue> *script* are individual actions that are executed in the order that they appear, much the way a computer program acts on lines of code.

OK. I've been trying to hide it, but scripts ARE lines of code – but designed to simplify the process for creating complex visualizations. This part of VisualEyes will be the hardest for most of you to grasp in doing your projects, but the payoff is big: With <glue> scripts, you will be able to do things easily in your projects that had to be custom-programmed by a computer programmer with years of experience.

Each line in a <glue> script contains a combination of glue methods and glue lists.

Glue Methods

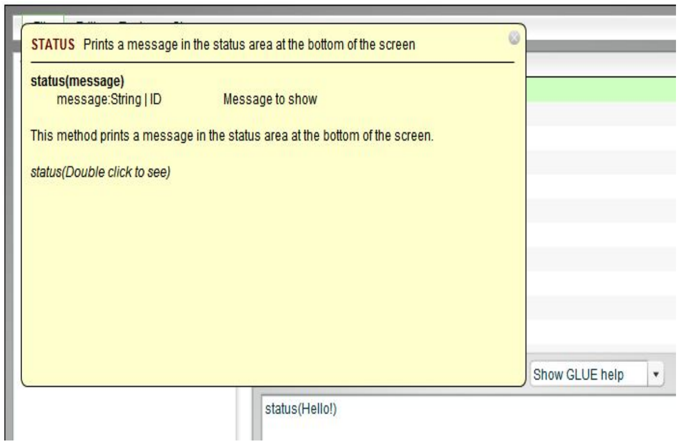
Methods are of built-in activities you can call upon to put in your glue scripts, such as:

- Running a query on a table of data
- Controlling a digital movie
- Animating items on the screen, or
- Calling up web pages

You can see a list of all of these activities in the appendix of this guide. These methods are also available to select from in VisEdit when you are editing a script in your project.

In a <glue> script, a method consists of the following:

- A name
- One or more parameters enclosed in parentheses


STATUS Prints a message in the status area at the bottom of the screen

status(message)
message:String | ID Message to show

This method prints a message in the status area at the bottom of the screen.

status(Double click to see)

status(Hello!)

Note that in VisualEyes, all of the activities or methods except for the list() method expect a certain number of *parameters*.

Parameters are bits of information the glue method needs to perform its function.

If more than one parameter is required by a method, they are separated by commas, i.e. add ($total,1,2).

As an example, one of the simplest methods is **status()**, which causes a banner-like message to appear at the bottom of the screen. For example, this script will print "Hello digital humanists!" whenever it is *called*, which in this case, is each time the screen is refreshed:

```
<glue init="true">
        status(Hello digital humanists!)
<glue>
```

Note that the <glue> did not need an *id*, since its *init* was set to "true" nor a *from*, since we aren't looking to show a resource, such as an image. When called, VisualEyes will look at each line between the start of the glue (**<glue init="true">**) and the end of the glue (**</glue>**) and run each line in the order it appears. In this case, just one line is involved.

1. The first line of this item instructs the VisualEyes to run the glue method each time the screen is refreshed.

2. The second line of this item is the *script*, and shows the name of the method (in this case, **status**) and one or more parameters enclosed in parentheses (**Hello digital humanists**!).

3. The third line of this item indicates the end of this method.

Lists

Understanding Variables/Lists

Whether it's a hangover from poorly taught 7th grade Algebra, or just a hard concept in its own right, the concept of variables is difficult for EVERYONE at first. It is an abstract way at looking at things that many people, especially humanists find foreign. In VisualEyes, a variable is called a **list.**

The bad news first: having a good idea of what we mean by variables in VisualEyes (we call them **lists**) is important for being able to make interesting VisualEyes projects. The good news is that this is a pretty simple concept to follow, if presented properly, and once you get it, everything else in VisualEyes will be easier.

Variables Defined

Variables are ways to describe a data element without having to say exactly what that something's value is. They are called variables because their contents can vary. They are used to take a concrete thing like a number, a word, or a list of words, and give it a name to call those items by.

This is useful so we can think about something like a *year* and not have say it is 1960 or 2010, so we can say things like, "if the *year* is 1980, show the picture with the big hair."

So, for example, in a VisualEyes project, a *year* could be a variable. But that *year* could change depending on the data you are working with or the data you want to display with that year. So you will want to create a list of all of the y*ears* you will want to use and to which you will want to associate your data. Hence, the *year* **varies** because it's a **variable.** As mentioned earlier, we call these variables **lists**.

Lists Are Containers

Here is another way to see how these **lists** work in managing and displaying data in VisualEyes:

- Imagine an office with a wall of filing cabinets made up of many drawers.
- Each drawer has a label on it to identify the contents within the drawer.
- Each drawer can contain one or many items of different types.
- **Lists** are like drawers, because they contain items we find by looking at the name we gave the drawer.

Lists are named containers that hold many kinds of items

1000
.23
Ask not, what your country can do for you, what you can do for your country.
www.viseyes.org
Name

Just as a drawer can contain papers, envelopes, and photographs, **lists** are containers because they hold one or more things we want to save, such as a number, a list of names, a URL, or any combination of these.

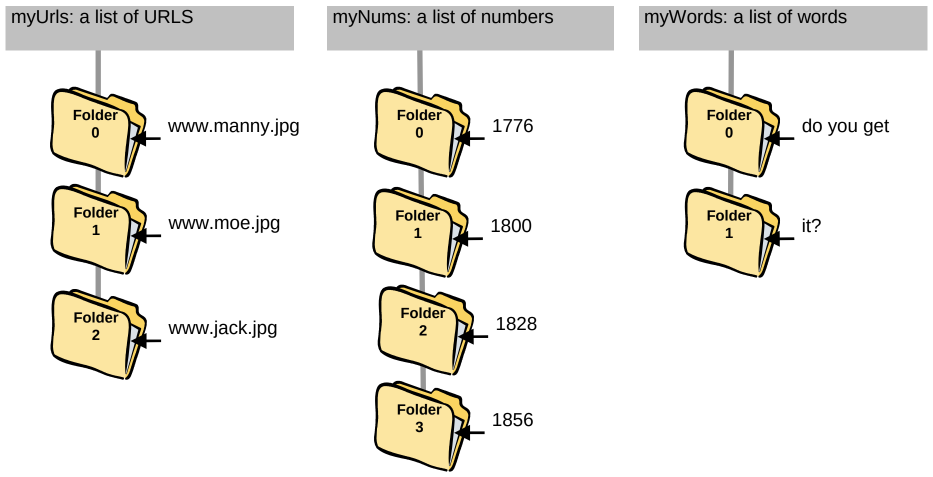
If there was to be just one drawer, we wouldn't need to label each drawer – but we can have a number of drawers. To find the drawer we want, we make up a name to uniquely identify the drawer.

Just like naming two drawers with the same name would be confusing, naming two **lists** the same name would make it hard to know which one we were talking about, so the names of each **list** should be unique.

Lists are named containers that hold many kinds and number of items

To further stretch the “drawers” analogy, the individual items in a drawer are in folders, numbered from 0 to however many items are in the drawer. Computers start numbering their lists at zero rather than one, so the 1st folder is labeled 0, the 2nd labels 1, and so on. For example, here are three lists

We find an item in the drawer by telling the drawer's label and the number of the folder, such as the 5th folder in the drawer called *myDrawer*. Needless to say, it makes sense to put related item in the same drawer.



VisualEyes Lists

When you write glue for your project scripts in VisualEyes, you will use two types of **lists** within <glue> items:

- **Global lists** are set by the program to respond when you click on a screen element to run an animation, for example, or to move the timeline. In a glue script, global lists are prefaced with two dollar signs ($$).

- **Local lists** are those that you create yourself to use temporarily to figure out a date to correlate with a data display, or to join some words together In a glue script, local lists are prefaced with one dollar sign ($).

Using Global Lists

Global list values are automatically set by the VisualEyes application in response to user actions like clicking on map area or moving the timeline slider, and are available to all glue items in the view.

For example, if you wanted to display the current year below the screen as you moved along the timeline, the <glue> script would look like this:

```
<glue init="true">
   status($$curYear)
</glue>
```

This script uses the **status()** glue method to display the current year on the screen.
In the second line of this script, you are calling on the global list, $$curYear. This global list, $$curYear, contains several dates. The dates will be called upon to change based on where the user moves along the timeline.

In drawer-speak, VisualEyes has created a drawer and labeled it $$curYear. Each time the timeline is moved, the item in that drawer, in this case a number representing the current year, is called upon. When the glue item runs, the **status()** method looks in that drawer called $$curYear, pulls out the item within it and writes it on the screen.

Global Lists in VisualEyes

There are a number of global lists that are useful to see what time the timeline is at and what dot or map feature was clicked on:

$$click	Gives the feature index of the currently clicked on map feature
$$param	Gives the index of the currently clicked on a dot
$$now	The time in the timeline from 0-1
$$curYear	The current year in the timeline
$$curMonth	The current month in the timeline expressed as mo/year
$$curDays	The current date in the timeline expressed as days +/- 1970

Using Local Lists

If we wanted to move the timeline with the mouse, and rather than display the year we were over, we wanted to add 10 years to the display (i.e. 1970 would show as 1980), we would need to make our own local list:

```
<glue init="true">
   add($myYear,$$curYear,10)
   status($myYear)
</glue>
```

Here we used the **add()** <glue> method to create a local list called **$myYear**, and set its value to the timeline's year **($$curYear**) plus **10**.

In drawer-speak, we have created a new drawer labeled **$myYear**. When the glue item runs, the **add()** method takes the following actions:

1. Looks in that drawer called **$$curYear**
2. Pulls out the value of the item in the drawer
3. Adds **10** to the value of the item in the drawer
4. Looks in the drawer called **$myYear**
5. Sets the item in that drawer to the value.

In addition, the **status()** method:

1. Looks in the drawer called **$myYear**
2. Pulls out the item
3. Writes it on the screen

Using Lists with many items

The examples of lists we've used so far only had one item in them, but as the name implies, lists can contain any number of items within them. Being able to include many items is very convenient, as we can create a script to talk about a lot of items without having to make a separate list for each one. For example, we could make a list containing the days of the week like this:

```
<glue init="true">
   list($days,Mon,Tue,Wed,Thu,Fri)
   status($days.1)
</glue>
```

In drawer-speak, when the glue item runs the **list()** method, the following actions take place:

1. The **list()** method creates a drawer labeled **$days**
2. The **list()** method adds 5 new folders to the drawer: Mon, Tue, Wed, Thu, and Fri , respectively. The word **Mon** placed in the 1st, **Tue** in the 2nd, etc.
3. The **status()** method looks in that drawer called **$days**
4. The **status()** method pulls out the 2nd item within it (in this case, **Tue***)*
5. The **status()** method writes **Tue** on the screen./ Remember that computers start numbering their lists at zero rather than one.

Commenting out lines

You can comment out lines of glue script by using /* */ to bracket the area, like this:

```
list($myData1,1,1,2,3,4,5,6,7,8,9)
/* list($myData3,9,9,9,2,2,9,9,9,9,9)
dataset(myGraph,1,Set two,$myData2)   */
dataset(myGraph,2,Set three,$myData3)
```

Or use // to comment from that point to the end of the line. This is useful for documenting the script:

```
list($myData1,1,1,2,3,4,5,6,7,8,9)                  // Data set 1
list($myData2,38,20,37,22,27,30,32,3,36,40)         // Data set 2
// list($myData3,9,9,9,2,2,9,9,9,9,9)               // Commented out
```

Tables

Accessing individual data elements in a table

Tables are typically accessed by querying the data with a query() method, but you can access individual elements by specifying them by field. For example**,** if we had a resource with the *id* of "myTable" and a field called "name", **status(*myTable.name)** would print a list of all the rows of the *name* field on the screen and **status(*myTable.name.1)** would print the 2nd name (the count starts at zero).

Working With Structured Information

Structured Information

One of the things that makes VisualEyes particularly useful is its ability to manage, ask questions of, and display items from large collections of ***structured*** information. Structured means that instead of the data and information presented like a text document, particular kinds of data are grouped together in meaningful groups.

In contrast, unstructured information is like a Word document file. All the information is put together with nothing separating the important elements like this box.

Bob is a 22 year old man who got 100 on the test.
Ted is a 43 year old man who got 40 on the test.
Carol is a 33 year old woman who got 90 on the test.
Alice is a 23 year old woman who got 75 on the test.

This may work for small amounts of information, but imagine needing to find all the men that passed the test if there were 400 people in the class. It would be very difficult to automate, as the computer would have a hard time telling the ages from the grades.

One solution is to *structure* the data. That is, if we know an item is a grade, put it in the "grade" group, while an item that is a name would go into the "name" group. We call these collections of structured information ***tables**,* and they are really no different than an ordinary Excel spreadsheet.

name	**sex**	**age**	**grade**	**class**
Bob	male	22	100	1
Ted	male	43	40	2
Carol	female	33	90	1
Alice	female	23	75	2

Both a spreadsheet and a table consist of multiple rows of information, sorted into useful groupings in columns. An example of this shown in the box to the left.

Each column is a grouping of related things, called a ***field***. The first row defines the names of the fields, followed by any number of rows that contain the information for the fields. In this table, there are 4 people (Bob, Ted, Carol, and Alice) and 5 fields (name, sex, age, and grade, and class).

Having the data structured makes it easier to make sense out of the table, and ask it better questions. Google is essentially an unstructured table of the web. When we search, it looks to see if any of our search words appear in a web page, and return those pages if it does. An example of an unstructured search would be if you searched for “Tiger” Woods, the golfer. Your search results would include data about the golfer as well as about the feline predator. By comparison, a structured search would involve searching specified fields in a structured table. For example, to conduct a structured search for the occurrences of “Tiger” as a name, you would indicate that you are searching for matches in a structured table’s *name* field." Structure adds a new level semantic meaning to our searches.

On the web, tables are stored in programs called databases. VisualEyes has a simple database built-in to support you in structuring your data into tables, and then to easily and quickly search for the portions of information you want from your tables.

Putting Your Table Online

The VisEdit webapp has a feature that will take a spreadsheet file, convert it to XML and store it on the VisualEyes server for you. As a result, you can use Excel (or just about any spreadsheet application) to import data to the VisualEyes server so that you can access it online. What follows are the steps to convert your data to XML:

1. **Open and format your spreadsheet.** To begin converting your data, the first row of your spreadsheet should contain a list of single-word field names by which each column can be referred (i.e. *name, sex, age, grade, id* in the previous example). The rows that follow within each column can contain any number of items of data sorted across the horizontal fields.

2. **Save your spreadsheet** using the CSV (comma delimited values) or tab-delimited text file formats available from most spreadsheets and database programs. To do this, use the "Save As…" option in the File menu and set the "Save as type" option to CSV (Comma delimited) or "Text- (Tab delimited)" and save to a file.

3. **Go to the VisEdit Editor**

   In the *Tools* menu of the VisEdit editor, select the *Convert Data File to XML* option:

   - A file dialog will prompt you to locate on your hard drive, the spreadsheet you want to convert.
   - Once selected, your spreadsheet will appear in the screen.
   - Make whatever changes you need to the raw text in this view, such as editing the field names on the first line so that they do not contain any spaces. We like to use *camelCase* (first letter of words in caps, except the first. i.e. myFirstName, myAge, etc.) as it makes for a pronounceable field names.

   You can load CSV files directly to the server using the "Upload CSV file to server" option in the *Tools* menu and it will be saved to server's data folder with your user number its name (i.e. *http://www.viseyes.org/data/1-BobTed.csv*). You can upload revised versions over this at any time if your data changes.

   NOTE: This will add 2-5 seconds to the load time when the project is run, so when you have finished making changes to the file for a while, save it as a native XML file for faster loading:

   > Click on the *Convert to XML* button to convert your spreadsheet data to XML format. You will be asked to type a name for the table to store it under. Click on the *Upload to server* button to save that file on VisualEyes server's data folder with your user number and the name you gave it (i.e. *http://www.viseyes.org/data/1-BobTed.xml*).

Querying a Table

The process of "asking" a table for certain data is called ***querying***. You do queries all the time on the web when you conduct a search. For example, when you try to find a movie in Netflix, you ask the Netflix server to search its table of movies by matching the words you typed in. Behind the scenes, your search words are sent to the server at Netflix, which "asks" the database to look through the genre you are in and return the titles of any films in which all your search words can be found. After a few seconds, Netflix displays a list of search results. The same process occurs when you search for books at the library website, Google, and even Apple's iTunes, which is no more than a simple database.

The Parts of a Query

To conduct a query in VisualEyes, you need three basic pieces of information to get the data you want from a given table:

1. The name of the table
2. The conditions
3. The desired fields from the source if the conditions are met

1. **The name of the table**
   The name of the table that contains the raw information you want to pick and choose from. Since any given project might have many tables, to choose from, you need to specify one of them by giving its name.

2. **The conditions**
   The conditions that need to be met before any rows are retrieved from the source table. Conditions are statements like, "all the people who scored below 70" but in a form that the computer can understand, such as "grade LT 70". We take advantage of the structured nature of our data and look at the "grade" field to return only people who have grades less than (LT) 70.

   A single condition like "grade LT 70" is called a *clause*. Each clause is said to be *true* if the condition is met (i.e. the grade is 50) or *false* if the condition is not met (i.e. the grade is 80).

   Each clause had three parts:

   1) the field to look at
   2) the conditional (i.e. GT, LT, EQ ...), and
   3) the value to compare with, which can be a number, word, sentence, or another field name.

   The conditions can get more specific by adding multiple clauses like any Boolean search. In our example, "men who scored over 60 and are under 40" is a condition that translates into three clauses joined by "sex EQ male" AND "grade GT 60" AND "age LT 40." The **AND** that separates each clause is called an *operator* and says "return rows if both the clauses it is between are *true*." Alternatively, we could use the **OR** operator which says "return rows if either of the clauses it is between are *true*."

3. **Which fields to return**
   Your table might have 5 fields, but you may only need to get one, such as the "name". To do this, you need to specify which fields to include in the results. Specifying "name" will return just the name (i.e. Bob), and "name+age" will return the name and age (i.e. Bob, 22). If you want all the fields, use a star ("*") (i.e. Bob,male,22,100,0).

Queries in VisualEyes

VisualEyes allows you to query locally without needing to send a request to a server. This is a big advantage in terms of performance over traditional web queries. In the Netflix example, we had to send a message via the Internet to the Netflix server, where it searched its database and returned the results back to us in a message. The query process we use in VisualEyes is modeled after the standard Boolean queries done by most commercial databases such as SQL, just simplified.

Queries are done using the query() method in a <glue> item. Just as was outlined earlier, a query() has three basic parts: the *source* table; the *conditions*; and the desired fields from the source if the *conditions* are me -- plus the name of a list to put the results in and how they are ordered.

The form of query is **query(resultsID, tableID, fields, conditions, orderBy)**, where the results of the query are returned in a *resultsID* from a table (*tableID*) consisting of the *fields* and rows meeting certain *conditions*, ordered by a field name (*orderBy*).

1. **The name of the table**
   This is the id of the <resource> that holds the XML table. Assuming we wanted to load the example we've been working with, you would add a resource to your view something like this:

   ```
   <resource id="myData" src="http://www.viseyes.org/data/1-BobTed.xml>
   ```

   Which assigns the name "myData" to the data loaded from the url "http://www.viseyes.org/data/1-BobTed.xml", making "myData" is the *tableID* for the query().

2. **The conditions**
   The *conditions* determine what rows will be included and contains one or more conditional clause. Each clause consists of a field name, a condition, and a value (i.e. name EQ Bob, age LT 30, etc.). Putting a * in the *conditions* place will cause all the data in the table to be sent to the list.

   There are the following *conditionals* possible:

EQ	Field is exactly equal to value
NE	Field is not equal to value
LK	Field contains the value with its string (like)
NL	Field does not contain the value with its string (not like)
LT	Field is less than to value
GT	Field is greater than the value
LE	Field is less than or equal to value
GE	Field is greater than or equal to the value

   The LK (like) conditional is a "fuzzier" search, used to find the occurrence of a word in an item, regardless of case. For example, "name LK bo" would return Bob's row.

   If the field contains multiple values, separated by a ; (semi-colon), each value will be searched and items that match will be included in the search results. For example, if Bob was in both classes, the class field would be "1;2", and our condition looked for people in class 1 (i.e. "class EQ 1") , Bob's row would be included in the results.

   For example, if we wanted to know all the people who scored below 70, the conditions would be "grade LT 70". Individual clauses may be joined by AND or OR operators to create more sophisticated queries, such as "grade GE 70 AND sex EQ men" if we wanted to know all the men who scored greater than or equal to 70.

3. **Which fields to return**
   To specify which fields within a row are added to the results, set an individual field name (ie. "name"), two or more fields, separated by a + sign (i.e. "name+age"), or a * (star), which will return all the fields on rows where the conditions are met.

4. **List to hold the results**
   We need a place to put the results of our query. The *resultsID* can be an existing list, or query() will create one if it doesn't exist. We would then use this list to fill an information box, or any other data display option.

   If you are only looking for one field to be returned (e.g. field="name") all items matching your conditions will be returned in the list (e.g. "Bob,Alice"). If multiple fields are selected (e.g. field="name+age"), only the first match is chosen and all the desired fields in that match are returned (e.g. field="Bob,22").

5. **What order**
   Finally, you can specify what order the rows are placed in the list by specifying the name of the field to order them in ascending order. Putting a 0 in will not order them.

Some Query Examples

name	sex	age	grade	class
Bob	male	22	100	1
Ted	male	43	40	2
Carol	female	33	90	1
Alice	female	23	75	2

Using this simple table, called "myData", let's work out some queries to pull out some specific items from it.

All examples assume we will place their results in a list called $results, order the results by class and use the following resource to load the table from the VisualEyes server (it's online if you want to try it yourself).

```
<resource id="myData" src="http://www.viseyes.org/data/1-BobTed.xml>
```

- **Find all Males**

```
query($results,myData,name,sex EQ male,class)
status($results)
```

Results are: Bob,Ted

- **Find all people younger than 40**

```
query($results,myData,name,age LT 40,class)
status($results)
```

Results are: Bob,Carol,Alice

- **Find a man older than 40 that passed**

```
query($results,myData,name+age+score,sex EQ male AND age GT 40 AND score GT 70,class)
status($results)
```

Results are: Bob,22,100

- **Find all people that have an "o" in their name**

```
query($results,myData,name,name LK 0,class)
status($results)
```

Results are: Bob,Carol

Table Glossary

Clause	A condition that must be met if an item is to be included
Conditional	A comparison such as less than, greater than, equals, etc.
Table	A structured collection of items organized into fields
False	The result of a clause that makes it omitted from in the results
Field	A category that an item is separated by type
Item	A line of information separated by fields
Operator	Used to join clauses together
Query	A request for a subset of table items meeting certain conditions
Results	A list of items that met the conditions posed
Structured	Information is sorted into fields
True	The result of a clause that makes it included in the results

Glue Methods

List Management

LIST

This method will create an array of elements (numbers, colors, or strings) under a named id for use in other methods. It can also create an array with only 1 element, for use as a variable.

list(listID, element1, element2, … elementN)

listID:String	Name of list
element:[number | color | string] …	List element(s)

The first example below will create a list of four numbers and makes that list available to other methods under the ID name called *$years,* with 3 years, an id to a 4th located in *$id* and a 5th provided by the global list *$$param*. The second example creates a list with a single element (a web site address) under the id *url*.

```
list($years,1865,1866,1877,$id,$$param)
list($url,www.primaryaccess.org)
```

COPY

This method will copy a member or members from one resource or list to another. If the destID is prefaced with "$$", a global list will be created if it doesn't already exist, whose scope is beyond the current calculation script.

copy(destID, destStart)

destID:String	Name of list to or resource to copy to
sourceID:String	Name of list to or resource to copy from

SELECT

This method selects one member of a *source* list based on the first member of a *which* list and places it in the *destination* list.

select(source, destination, which)

source:String	ID of list of values to select from
destination:String	ID of list where selection is placed
which:Number	ID or number of selection number

SEGMENT

This method will sort data into a number of preset categories and use those as criteria to create a new list.

segment(sourceID, destID, filters, values)

sourceID:String	ID of source data resource
destID:String	ID of destination data resource
filters:String	ID of list of numbers to segment data
values:String	ID of list of values to assign segmented data

As an example, suppose we wanted to color a map so that populations of different area are drawn in different colors. Areas with no people should be colored white, populations from 0-25 colored light red, 25-50 medium red, 50-75 red, and population greater than 75 colored bright red:

```
<resource id="myData" type="data"
src="http://mysite.com/pop1845.xml" />
<resource id="myMap" type="map" src="http://mysite.com/myMap.xml"
/>
<glue from="myMap" init="true">
      list($slots,0,25,50,75)
      list($colors,0xffffff,0x330000,0x990000,0xff0000)
      segment(myData.pop, myMap.col, $slots, $colors)
</glue>
```

The segment method can be used to figure out which slot a time period figures in. Whatever date the timeline (the property called *now*) is at is compared to the slots, and the mapNum list is set to number from 1-the number of slots. (i.e. 1855 = 1, 1860=2, 1890=4)

```
list($slots,1850,1860,1870,1880)
list($mapNum,0)
segment($$now, $mapNum, $slots, null)
```

LISTMERGE

This method will join all the entries in a list into one entry, with a spacer between each if set. destID will create a new list, if that list does not already exist.

listmerge(destID, srcID, spacer)

destID:String	Name of list to put combined entries
srcID:String	Name of list to join into one list entry
spacer:String	Value to between entries

LISTNUM

This method will count the members in srcID and place that number in destID. destID will create a new list, if that list does not already exist.

listnum(destID, srcID)

destID:String	Name of list to put combined entries
srcID:String	Name of list to join into one list entry

LISTSPLIT

This method will look at each member in srcID and if it contains the sep separator, split that into how many parts are there, adding the new members to the end of the list

listsplit(srcID ,sep)

srcID:String	Name of list to split members of
sep:String	Separator character or string

LISTFILL

This method sets any values in a list called destID whose index appears in a list called srcID to the value specified in matchVal. All those not specifically in srcID would be set to the default val.

listfill(destID, srcID, matchVal, defaultVal)

destID:String	Name of list to fill
srcID:String	Name of list to specify (Null == all)
matchVal:String	Value to set matching indices in destID to
defaultVal:String	Value to set all other indices in destID to

```
        list($mainList,a,b,c,d,e,f,g)
        list($index,1,4,6)
        listfill($mainList,$index,yes,no)
        replaceword(myBox,words)
</glue>
```

Will result in a $mainList of this: *no,yes,no,no,yes,no,yes*

LISTJOIN

This method will join the second list to the end of the first.

listjoin(firstID, secondID)

firstID:String	Name of to join and hold both
secondID:String	Name of list to add to first

SET

This method copies srcID and places the result in the list or resource called destID.

set(destID, srcID)

destID:String	Name of list or resource member
srcID:String	Name of list, literal or resource member

LOOKUP

This method will return the nth member of a list or dataset within a resource.
destID will create a new list, if that list does not already exist.

lookup(destID, srcID, index)

destID:String	Name of list to put member
srcID:String	Name of list to or resource
index:Number	Index of srcID member to get

TWEENLIST

This method will set a list to tween (animate between to values) between two other lists over time. Useful when animating values of charts and graphs

tweenlist(destList, fromList, toList, percent, eases)

destList:String	ID of tweened list
fromList:Number	ID of from values list
toList:Number	ID of to values list
percent:Number	Percent of tween from 0-1
eases:Number	slows (0=none1=start 2=end 3=both)

SPLIT

This method will split a string into a list of separate parts by a letter or group of letters.

split(destID, num1ID, num2ID)

destID:String	Name of list to copy to
sourceID:String	String to split up by separator
separator:String	Letter(s) that separate parts

```
<glue id="split-em">
      split($parts,first&second&third)
      status($parts)
</glue>
```

Will result in: *first, second, third*.

JOIN

This method combine 4 separate parts into one element of a list. Leave blank extra sources, but keep number of parameter the same (i.e. split($list,a,b,,)

split(destID, source1ID, source1ID, source1ID, source1ID)

destID:String	Name of list to copy to
source1ID:String	String to be joined
source2ID:String	String to be joined
source3ID:String	String to be joined
source4ID:String	String to be joined

DATETODAYS

This method will convert a date expressed as a year, month/year, or day/month year (separators can be \ - : / or ;) into a single number representing the number of days +/- of January 1, 1970. For example, 1/1/1980 would convert to 3650 and 1/1/1960 would be -3650.

datetodays(daysID, dateID)

daysID:String	Name of list to put days into
dateID:String	Date to convert

DAYSTODATE

This method will convert the number of days +/- of January 1, 1970 into a readable date in the the form described by *format* (dy/mo/yr, mo/yr, yr, mo/dy/yr).

daystodate (dateID, daysID, format)

dateID:String	Name of list to put date into
daysID:String	Days to convert
format:String	Format for date

REPLACEWORD

This method looks at some text and replaces special symbols with a word or words. The symbols such as $$1, $$2, etc., where the $$ identifies it as a symbol and the number following it says which one in the list it should be replaced with. The replacement parameter is the ID of a list of replacement word or words. $$1 would be replaced by the first member in the list, $$2 would replace the second member, etc.

replaceword(textID, replacements)

textID:String	ID of resource containing text with symbols
replacements:String	ID of list of values to replace symbols with

```
<resource id="myBox" type="infobox" position="north">
      This is the $$1, this is the $$2, and this is the $$3.
      <frame wid="200" hgt="150" " backCol="0xffffcc" />
</resource>
<glue from="myBox" init=”true”>
      list($words,first,second,third)
      replaceword(myBox,$words)
</glue>
```

Will result in: *This is the first, this is the second, and this is the third*.

Math

ADD

This method adds numbers in num1ID and num2ID and places the result in the list called destID (i.e destID=num1ID+num2ID).

add(destID, num1ID, num2ID)

destID:String	Name of list to copy to
num1ID:String	Name of list, literal or resource member
num2ID:String	Name of list, literal or resource member

SUB

This method subtracts num1ID from num2ID and places the result in the list called destID (i.e destID=num1ID-um2ID).

sub(destID, num1ID, num2ID)

destID:String	Name of list to copy to
num1ID:String	Name of list, literal or resource member
num2D:String	Name of list, literal or resource member

MUL

This method multiplies num1ID by num2ID and places the result in the list called destID (i.e destID=num1ID*um2ID).

mul(destID, num1ID, num2ID)

destID:String	Name of list to copy to
num1ID:String	Name of list, literal or resource member
num2D:String	Name of list, literal or resource member

DIV

This method divides num1ID by num2ID and places the result in the list called destID (i.e destID=num1ID/num2ID).

div(destID, num1ID, num2ID)

destID:String	Name of list to copy to
num1ID:String	Name of list, literal or resource member
num2ID:String	Name of list, literal or resource member

INC

This method adds one to num1ID and places the result in the list called destID (i.e destID=num1ID+1).

inc(destID, num1ID)

destID:String	Name of list to copy to
num1ID:String	Name of list, literal or resource member

MIN

This method copies the smallest of two numbers and places the result in the list called destID (i.e destID=min(num1ID,num2ID).

min(destID, num1ID, num2ID)

destID:String	Name of list to copy to
num1ID:String	Name of list, literal or resource member
num2ID:String	Name of list, literal or resource member

MAX

This method copies the largest of two numbers and places the result in the list called destID (i.e destID=max(num1ID,num2ID).

max(destID, num1ID, num2ID)

destID:String	Name of list to copy to
num1ID:String	Name of list, literal or resource member
num2D:String	Name of list, literal or resource member

ABS

This method copies the absolute value of a number and places the result in the list called destID (i.e destID=abs(num1ID).

abs(destID, num1ID)

destID:String	Name of list to copy to
num1ID:String	Name of list, literal or resource member

FLOOR / ROUND

The floor method returns the integral portion of num2ID and places the result in the list called destID. The round method returns the rounded value of num2ID and places the result in the list called destID.

floor(dest, num1ID) -- and -- round(dest, num1ID)

destID:String	Name of list to copy to
num1ID:String	Name of list, literal or resource member

Logic

IF

This method will execute the number of lines specified if condition between var12 and var2 is met. *Note: There is NO SPACE between "if" and "("*

if(var1, condition, var2,numLines)

var1:String	Name of list, literal or resource member
condition:String	Condition (GT, LT, EQ NE, LE GE,LK,NL)
var2:String	Name of list, literal or resource member

REPEAT

This method will repeat the script lines between the first time it is called with a number (the number of times to repeat) and the second time it is called with 'end' as its parameter (no quotes!). Useful for looping things, as a tradition do or for loop

repeat(numID)

numID:String	Name of list, literal or resource member

For example, this script would show the number 4 on the screen:

```
list($count,0)
repeat(4)
    inc($count)
repeat(end)
status($count)
```

Data Management

QUERY

query(listID, dataID, fields, conditions, orderBy)

listID:String	ID of list results are placed
dataID:String	ID of resource where row/col data is
fields:String	Fields to include separated by a +
conditions:String	Inclusion conditions separated AND or OR
orderBy:String	Field to order row results by (0=none)

```
query($myList,myData,*,year EQ 1847 AND county EQ LA)
```

Would return all the fields from myData, where the year was 1847 and the county was LA into a new list called myList.

```
query($myList,myData,year+county,year GT 1847 OR county NE LA)
```

Would return just the year and county field from myData, where the year was greater than 1847 or the county was not LA into a new list called myList.

DATASET

This method adds a row of data to a graph.

dataset(graphID, set, legend, data)

graphID:String	ID of source graph resource
set:String	number of data set
legend:String	legeng
dataID:String	ID of data to add

DATATIME

This method sets the amount of data to show in a graph to make time series data appear with, such as following the position of a timeline. (use the $$now parameter to track timeline from 0-1). On line and area charts, the number of points shown on the x-axis will be mediated by the time set. For example, if a chart has 30 elements, setting time to .1 will show the first 3.

datatime(graphID, time)

graphID:String	ID of source graph resource
time:Number	Amount of data to show (0-1)

NORMALIZEGRAPH

This method will set the status of a graph set by *graphID* to plot the data as raw numbers by setting max to 0 (it's default condition) or normalize the data from 0 to the number set by *max*, typically 100. This is useful when trying to compare datasets with wildly different ranges.

normalizegraph(graphID, max)

graphID:String	ID of source graph resource
max:Number	Maximum number on Y axis

FILLDOCVIEWER

This method will fill a document viewer object with data from a data source (i.e. A an XML file, or a SQL database query). You can select a specific item in the data source by setting the title parameter in the glue call to the item's number prefaced with a # sign (i.e. #32).

filldocviewer(viewerID, title, dataID)

viewerID:String	ID of docviewer resource
title:String	Title to look for in data
dataID:String	ID of data resource

DOTFILL

This method will fill a container object, such as a path or concept with dot data from a data source (i.e. A an XML file, or a SQL database query). The data source must contain at least the x,y, and time fields. See the dot specification for more information.

dotfill(containerID, dataID)

containerID:String	ID of container to house dots
dataID:String	ID of data resource

ROUTEFILL

This method will fill a container object, such as a path or concept with route data from a data source (i.e. A an XML file, or a SQL database query). The data source must contain at least the start, end, and pathway fields. See the route specification for more information.

routefill(containerID, dataID)

containerID:String	ID of container to house routes
dataID:String	ID of data resource

All Else

MOVE

This method will move a resource over time. If the timing is set to *0*, the resource will always be positioned at the starting positions specified. An id of *screen* can be use to move entire screen.

move(resourceID, startX, startY, startZ, endX, endY, endZ, timing, eases)

resourceID:String	ID of resource or *screen*
startX:Number	starting horizontal position
startY:Number	starting vertical position
startZ:Number	starting zoom position
endX:Number	ending horizontal position
endY:Number	ending vertical position
endZ:Number	ending zoom position
timing:String	ID of timing source (i.e. timeline, var, 0)
eases:Number	slows (0=none1=start 2=end 3=both)

TWEEN

This method will set a resource field to some position over time. If the timing is set to *0*, the resource will always be positioned at the starting positions specified.

tween(fieldID, start, end, timing, eases)

fieldID:String	ID of field, with '.' modifiers
start:Number	starting value
end:Number	ending value
timing:String	ID of timing source (i.e. timeline, var, 0)
eases:Number	slows (0=none1=start 2=end 3=both)

LINK

This method will cause a webpage to open. The "http://" portion of the URL is not required. You can specify the name of a *list* method in place of a URL, in which case, the URL name can respond to a click, say from a path object. Target sets where the page will open, which can be set to the frame's name or the preset values of *_blank, _self, _parent,* or *_overlay* (which opens an iFrame over the screen area). The *clickParam* will cause the current click parameter (0 if none) to be appended to the url as ?id=# (or &id=# if there is a name=value pair already there.)

link(url, target, clickParam)

url:String	full URL of page to load, or ID of list
target:String	browser window or frame
clickParam:Boolean*	if set to *true*, ?id= will be added to url

When a map is clicked on, the feature number associated with the feature clicked on will be available to methods that support the *clickParam* option, such as the **link** method.

SHOW

This method sets the visibility of a resource. The resource can be rendered fully transparent (opacity=0) to fully opaque (opacity=100) or any point in between.

show(resourceID, opacity)

resourceID:String	ID of source data resource
opacity:Number	Opacity of resource 0-100

STATUS

This method prints a message in the status area at the bottom of the screen.

status(message)

message:String	ID of list with message, or literal

REFRESH

This method will cause the resource identified to be redrawn.

status(resourceID)

resourceID:String	ID of source data resource

DISSOLVE

This method will dissolve between two resources. Times are expressed as 0-1, with one being the length of the timeline and 0 its start.

dissolve(inID, outID, start, mid, end, dur)

inID:String	ID of incoming resource
outID:String	ID of outgoing resource
start:Number;	Start time of outgoing res (0-1)
mid:Number;	Start time of incoming res (0-1)
end:Number;	End time of incoming res (0-1)
dur:Number	Duration of dissolve transition (0-1)

RADIOSHOW

This method acts like a radio button, and sets the visibility of a list of resources such that only one is visible at any given time. The selected resource can be rendered fully transparent (*opacity*=0) to fully opaque (*opacity*=100) or any point in between. All others are hidden. Setting select to 0 hides them all. The *select* can also reference an ID of a list. When using *radioshow* to *select* between dot object, use the word “dot” as the *resources* list.

radioshow(select, opacity, resources)

select:Number	which resource 1-N
opacity:Number	Opacity of selected resource 0-100
resources:String	ID of list of resource IDs

GOTOTIME

This method will cause the timeline to go to the date specified in *when*, the number of days +/- of January 1, 1970.

gototime (when)

when:Number	When to go on the timeline

CALL

This method will run another glue item in the current view as a subroutine. Any parameters passed will be available in the $$param and $$click global lists.

call(glueID, params)

destID:String	Name of glue item to execute
param:String	Parameters to pass to glue

SETVIEW

Views can also be invisible and not associated with any particular tab. By setting the *visible* attribute to “true” and giving it an *id*, you can use GLUE to cause a view to show within the currently active tab’s screen space. If the view is a visible one, the view’s tab will be activated. See the **setview** GLUE element and the section on the cookbook section for more information on this very powerful option.

setview(viewID)

viewID:String	Name of view to show

MOVIE

This method will control a movie resource’s transport functions such as play or stop.

movie(resID, command, param)

resID:String	Name movie resource to control
command:String	Operation to do
param:String	Parameters to pass to command

Current movie commands are:

play	The param is the time in seconds to start playing the movie from
stop	The param is set to 0
seek	The param is set to the time in seconds to cue the movie to
time	The param the name of the list to store the current time in seconds
start	The param is the time in seconds of movie’s start time
end	The param is the time in seconds of movie’s end time
load	The param is the src/path of the movie to load

PLAY

play(startTime)

startTime:String	Time to start playing

This method will cause the timeline to play from the time specified in startTime. It is the same as if you dragged the timeline slider with the mouse and clicked the play button.

MENUITEM

This method will change an item in a control panel to a new title, glue or value. Leaving a parameter blank keeps the old value of it intact.

menuitem(controlID, title, glue, value)

controlID:String	ID of control panel item
title:String	Name of new title
glue:String	Name of new glue
value:String	Value of new item

VisualEyes XML Cookbook

The following topics will help you add specific elements common to many VisualEyes projects. Be sure to check the specific documentation described earlier for the elements involved. Using the VisEdit editor (www.viseyes.org/edit.htm) will help you get started by providing the basic elements, automatic XML formatting, and embedded help.

1. How to set up a basic project

The following example is the bare minimum required for a VisualEyes project, and causes it to be shown on the screen. If you are using the VisEdit editor, it will be provided automatically for new projects when you click on the “New” menu option.

```
<project title="My Project">
        <frame wid="800" hgt="500" frameWid="1" />
        <textformat size="11" font="_sans" />
        <tab hgt=”15” wid=”150” />
        <view title="My View" />
</project>
```

The first line creates an element for the entire project. The *title* attribute is internal and not shown in the project. The second line is a **frame** element that defines the basic shape for the views in the project. A **textformat** element can also be added to control how the text will appear. See the textformat documentation for the various options available. All other elements “inherit” this format and use it as the basis their text elements will be drawn. A **tab** element sets the size and look for the tabs that appear above the various **view** elements, where the content is shown.

2. How to add a base map

The following example adds a jpeg file to the project, and causes it to be shown on the screen. This must be added within whatever view you want the image to appear.

```
<resource id=”baseMap” src=http://mySite.org/myPic.jpg />
<glue from=”baseMap” init=”true” />
```

The first line creates a new resource called *baseMap* from the web-accessible jpeg file called *myPic.jpg* on the *mySite.org* website. Just because you have created a resource, you wont be able to see it until it is glued to the screen. The second line causes the resource named *baseMap* to be visible every time the panel is refreshed by setting init to *true*.

3. How to add a zoom control

The following example adds a zoom controller to the project. This controller will zoom in and out the entire screen, in a similar way that the controller on Google maps operates. This must be added within whatever view you want the zoom control to appear

```
<zoomControl top="24" left="30 hgt="80" max="5" />
```

The *top*, *left*, and *hgt* attributes set it’s position and size on the screen, in terms of pixels., The *max* attributes how many times you will be zoomed in when the scroller control is at the top (i.e. 5 = 500%).

4. How to add a timeline

The following example adds a timeline controller to the project. This controller will allow you to set a time that other elements can work with, as well as can add a player button to play animated progressions through time. This must be added within whatever view you want the timeline to appear

```
<timeline min="4/1954" max="1994" play="true" >
  <textformat col="0x999999" size="11" font="_sans" />
  <frame wid="550" hgt="550" left="120" top="530" backCol="0x999999" />
</timeline>
```

The timeline element contains a **frame** element, its *top*, *left*, wid, and *hgt* attributes set it’s position and size on the screen, in terms of pixels. A **textformat** element can also be added to control how the timeline text that displays dates will appear.

The *min* and *max* attributes how set the bounds of the time span represented. Setting the play attribute will add a player button. See the section on Timelines in this guide for all of the various options to customize a timeline.

5. How create an XML Data File

The easiest way to import data into VisualEyes is using Excel to create a spreadsheet. The top line should contain the field names and the following lines that data for those fields. For example, this format defines 3 fields, name, sex, age and has 4 people's information:

```
name                sex           age
bob                 male          22
ted                 male          43
carol               female        33
alice               female        23
```

Save this out as a tab-delimited text file in Excel by selecting "Save As…" in the File menu and setting the "Save as type" option to "Text- (Tab delimitated)" and saving to a file. The VisEdit editor has a tool that will allow you to load that file from your computer to the screen area, where it will be formatted into an XML format like this:

```
<TABLE a="name" b="sex" c="age">
       <ROW a="bob" b="male" c="22" />
       <ROW a="ted" b="male" c="43" />
       <ROW a="carol" b="female" c="33" />
       <ROW a="alice" b="female" c="23" />
</TABLE>
```

Click on the "Upload to server" button and that file will be saved on VisualEyes server's data folder using your user id and a name you gave it (i.e*. data/1234-myXMLFile.xml*).

6. How load an XML Data File

An XML data set is loaded into a project's view by way of a **resource** element. That element specifies where the file is (it's *src*), and an *id* to refer to it by when it is loaded from the server.

```
<resource id="myData" type="xml" src="data/1234-myXMLFile.xml"/>
```

Will load a file called from VisualEyes server created in the previous cookbook recipe. (If *src* was set to "http://mysite.org/data.xml", it would load a file called data.xml on the server at mysite.org). That file can have any number of fields and rows.

The data is now accessible to be queried and displayed by its id name. You can access individual elements by specifying them. For example**, `status(*myData.name)`** would print *"bob,ted,carol,alice*" on the screen and **`status(*myData.name.1)`** would print *"ted"* on the screen, since ted is the 2nd name (the count starts at zero).

7. How to create an animated path

Creating an animated path like animation of Jefferson's letters in the Jefferson's Travels project (http://www.jeffersonstravels.org) is relatively easy. You create a project with a base map and a timeline, then add a **path** to it that contains **dots** that specify that stops. The following example will create a path called *myPath* that responds to changes in time from the timeline by moving an icon between the carious dot positions and drawing a red line as it goes.

```
<path id="myPath” wid="10" headStyle="icon:letter" col="0xff0000" tweenLines="true">
        <dot date="1839" x="601"  y="168" />
        <dot date="1840” x="1034" y="1083"/>
        <dot date="1841" x="1759" y="959" />
</path>
<glue from="myPath" init="true" />
```

The path element draws a 10 pixels line set by *wid*, and that line is preceded by an icon defined by the *headStyle* attribute. Setting *tweenLines* to true causes a partial line to be drawn between the dots.

Within the **path** element are 3 **dot** elements that actually define the path’s course in terms of pixels on the screen. You can find the x and y values by clicking your mouse over the image in the preview and the values will be shown in the bottom-right corner. These numbers will not change if you zoom in and are based on the height and width of the base image.

The *date* attribute determines when that dot will be reached, and responds to the timeline’s position as when to be drawn. This can be expressed as year, month/year, or day/month/year.

The final line is a glue element to cause the path to be drawn on the screen. Setting *init* to true will cause the path to be redrawn each time the screen is refreshed, either clicking on the tab or any change in the timeline.

8. Adding Dots via XML to Paths

If your path has a lot of dots, it may be better to put the dots in an xml file and load them once rather than embedding them within the path item. Create an xml file that contains all the information about the dots as if you were embedding them in the path directly. The easiest way to do this is create an Excel spreadsheet with the data.

The first line should contain the field names (lower case only!) and the following lines the dot information, such as this:

```
date          x      y      glue
1/1765        344    654    showInfo
1/1772        323    444    showInfo
1/1790        244    334    showInfo
…
```

Upload and convert the Excel file the server (See the appendix for more information about the process) and fill the path with the dots like this:

```
<resource id-“myData” type=”xml” src=”/1234-MyData.xml”/>
<glue init="true" once=”true”/>
        dotfill(myPath,myData)
</glue>
```

The first line creates and loads the xml file we created and uploaded. The glue is set to run only once and uses the **dotfill()** method to fill the path called myPath, (which has no dots assigned to it yet) with the data from the mData resource.

9. Using routes in animated paths

If you have a number of journeys along a set number of path ways, you can define a collection of dots as a **pathway** and then use that pathway repeatedly at different times. In a normal animated path, you add a **path** element and add the **dots** for each leg of the path. To use a **route**, you set up the **path** element the same way, but instead of adding **dots** directly to the path, we set up an intermediary element called a **pathway**, and add the dots to it. This **pathway** is in turn called upon by a **route** element to be drawn to the screen.

The following script creates a path with two **pathways**, *toEngland*, which contains points on a map of mail to England, and *fromFrance*, which contains points on a map of mail from France. Notice that the dots contain percentages, instead of *times* or *dates* in the other direct method. *The* pct attributes can range from 0-1 and derive the dates from the **route** element later. The first dot is always 0 and the last dot always 1. In the toEngland example because the pct is .5, it will appear halfway and in fromFrance, the second dot will appear 20% into the route.

```
<path id="myPath" wid="10" headStyle="icon:letter" col="0xff0000" tweenLines="true">
        <pathway id=toEngland">
                <dot pct="0"  x="601"  y="168" />
                <dot pct=".5" x="1034" y="1083"/>
                <dot pct="1"  x="1759" y="959" />
        </pathway>
        <pathway id=fromFrance">
                <dot pct="0"  x="507"  y="156" />
                <dot pct=".2" x="934" y="989"/>
                <dot pct="1"  x="757" y="877" />
        </pathway>
        <route pathway="toEngland" start="1/2/1786" end="1/12/1786=" />
        <route pathway="toEngland" start="1/2/1786" end="1/22/1786=" />
        <route pathway="fromFrance" start="3/2/1786" end="3/12/1786=" />
</path>
<glue from="myPath" init="true" />
```

The route element is where it actually is drawn. The *pathway* attribute defines what **pathway** to draw and *start* and *end* attributes define when the whole pathway is drawn. The **glue** element causes the path to be shown on the screen.

10. Adding Routes via XML to Paths

If your path has a lot of routes, it may be better to put the routes in an xml file and load them once rather than embedding them within the path item. Create an xml file that contains all the information about the routes as if you were embedding them in the path directly. The easiest way to do this is create an Excel spreadsheet with the data.

The first line should contain the field names (lower case only!) followed by the route information:

```
start           end             pathway
1/20/1865       2/10/1865       Chicago
1/20/1865       2/10/1865       Detroit
```

Upload and convert the Excel file the server (See the appendix for more information about the process) and fill the path with the routes like this:

```
<resource id-"myData" type="xml" src="/1234-MyData.xml"/>
<glue init="true" once="true"/>
        routefill(myPath,myData)
</glue>
```

The first line creates and loads the xml file we created and uploaded. The glue is set to run only once and uses the **routefill()** method to fill the path called myPath, (which has no routes assigned to it yet) with the data from the mData resource.

11. How to make booklets for a docviewer

Booklets are a useful way to present information in VisualEyes. Booklets resemble page spreads in traditional books. Booklets contain one or more pages. Each page can have a picture on the left-facing page and text on the right-facing page, a single picture, or text crossing both sides. Each page has the following information:

- **title:** The title of the page
- **src:** The full address of the image (i.e. http://www.mySite.org/myPic.jpg)
- **caption:** Caption to appear underneath the picture
- **desc:** The text with any markup you want to add (i.e. bold, italics, links)

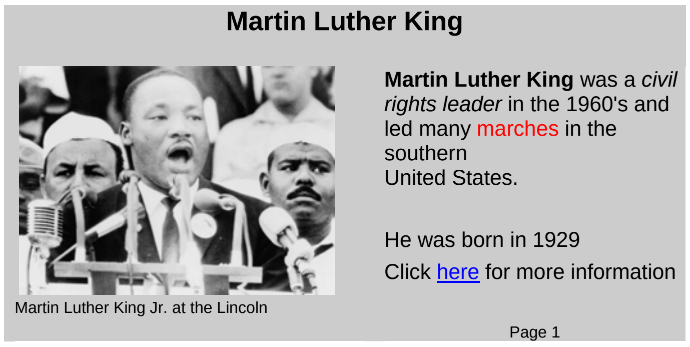
In the text you can use the following mark-up tags to control how the text will look:

- **b(text)** will bold the text within the parentheses
- **i(text)** will italicize the text within the parentheses
- **u(text)** will underline the text within the parentheses
- **font(face,size,color)** will size and color the text that follows
- **sp(leading,indent,tapstops)** will set the leading, indent & tabstops for text that follows
- **t()** will add a tab
- **br()** will add a line break
- **link(url,text**) will add a link to url when the text in the parentheses is clicked. If the url starts with *http://,* a new browser window will open and display the page, otherwise, VisualEyes assumes it is the name of a <glue> item and calls that <glue> item.
- **align(side)** will set the alignment for the text that follows. Can be *left, right* or *center*

For example a booklet title of “Martin Luther King” and the caption set to “Martin Luther King Jr. at the Lincoln Memorial”, and the following markup in the desc:

```
font(_sans,11,000000)b(Martin Luther King Jr.) was a i(civil rights leader)
in the 1960s and led many font(_sans,18,ff0000)marches
font(_sans,11,000000)in the southern United States. br()br()He was born in
1929.br()Click link(http://memory.loc.gov,here) for more information"
```

will yield a booklet that looks something like this:



Martin Luther King

Martin Luther King Jr. at the Lincoln

Martin Luther King was a *civil rights leader* in the 1960's and led many marches in the southern
United States.

He was born in 1929

Click here for more information

Page 1

The data source can have 4 fields: *title, source, desc* and *caption*. Make sure the field names are exactly spelled as those 4 and are in lower case. The *title* field provides a title at the top and a way to select items from the data source. Items with the same title will appear as pages within the document viewer. The *source* field gives a url for a picture if desired, and *desc* is an html formatted text area. If a *caption* field is defined, it will appear underneath the picture.

If both *desc* and *source* are defined, they will appear side by side. If only one is defined, only that one will appear. The text and picture information is supplied by the *filldocviewer* glue method, typically as the result of a query method. The best way to create booklets is by making an Excel spreadsheet where there are columns labeled **title**, **src**, **caption**, and **desc**, like this:

```
title         src                 caption       desc
Chapter One   http://myurl.com    pic1          The first page in chapter one
Chapter One   http://myurl.com    pic2          The second page in chapter one
Chapter One   http://myurl.com                  The third page in chapter one
Chapter Two   http://myurl.com    pic3          The first page in chapter two
...
```

A typical script would look like this, where the VisEdit editor was used to convert the tab-delimited txt file to XML and uploaded to the server:

```
<resource id="myData" type="xml" src="data/92-tobacc" />
<resource id="viewer" type="docviewer">
        <frame wid="600" hgt="400" left="150" top="50 />
</resource>
<glue id="showInfo" from="viewer" >
        filldocviewer(infoBox,*,myData)
</glue>
```

The 1st line loads that data into a table called “myData.” The 2nd line creates a docviewer called “viewer”. The 3rd line adds a frame to set the sizes and colors of the doc reader. The 5th line adds a glue element called “showInfo” that will display the docviewer on the screen and the 6th line fills the pages of the box with the entire contents of the data in myData.

12. How to create a menu using a *path* container

The following example will show 3 small jpeg files, each containing the chapter name on the screen at the position specified by the **x** and **y** tags. When any one of the jpegs is clicked, the Glue method called **chapterSelect** will be called, and the index of the dot (0-n) is stored in the global variable called **$$param**. The **radioshow** method then makes only that one visible by making its alpha 100% and all the unselected ones 0%.

```
<path id="chapters">
    <dot x="50" y="136" style="chap1.jpg” onclick="chapterSelect" />
    <dot x="50" y="164" style="chap2.jpg” />
    <dot x="50" y="192" style="chap2.jpg” />
</path>
<glue id="chapterSelect" from="chapters" init="true">
    radioshow($$param,100,dot)
    [ Do something you want here ]
</glue>
```

13. Making PopUp Information Boxes

Information boxes are popup boxes used to display textual information on demand. They are typically called by clicking on path and graph elements. InfoBoxes can contain a variant of HTML formatting and can be populated using search and replace variable that can be set using a database. The appendix contains detailed information on the text formatting options available.

An **infoBox** is a type of resource of *type* infoBox. It contains the text to be displayed followed by a frame tag that sets the size and look of the infoBox. Since infoBoxes typically popup after a mouse click, the *position* attribute determines the direction from the mouse the box is drawn. In this example, an infoBox called myBox is created that will display some text with two replaceable parameters, $$1 and $$2. Whenever the **glue** element *showBox* is called, the spot held by *$$1* will be replaced with *Memphis* and *$$2* with *New Orleans* for illustration purposes (Normally, these would dynamically come from a dataset of some sort).

```
<resource id="myBox" type="infobox" position="north">
        font(_sans,12,0x990000)May 6, 1831
        font(_sans,11,0x000000)sp(0,0,60)
        Shipped from: $$1
        To: $$2
        <frame backCol="0xFFFFCC" corner="6" wid="200" hgt="150" />
</resource>
<glue id="showBox"from="myBox">
        list($cities,Memphis,New Orleans)
        replacetext(myBox,$cities)
</glue>
```

14. How to Query Data

Being able to query data without needing to send a request to a server is a big advantage in terms of performance. There are 3 parts to querying data: 1). Defining the data set to be queried, 2). Specifying what parts of that data you want to get, and 3). Doing something with the results you get. The results can be ordered by any field and putting in a * in the query portion will result in all fields being returned.

The form of query is **query(listID, dataID, fields, conditions, orderBy)**, where the results of the query are returned in a *listID* from a data set (*dataID*) consisting of the *fields* and rows meeting certain *conditions*, ordered by a field name (*orderBy*).

The *listID* can be an existing list, or the query will create one if it doesn't exist. The *fields* can be an individual field, by name, two or more fields, separated by a + sign (i.e. "name+age"), or a *, which will return all the fields on rows where the conditions are met. The *conditions* determine what rows will be included and contains one or more conditional clause. Each clause consists of a field name, a condition, and a value. (i.e. name EQ John, age LT 20, etc.). Putting a * in the *conditions* place will cause all the data in the data set to be sent to the list.

There are the following conditions possible:

EQ	Field is exactly equal to value
NE	Field is not equal to value
LK	Field contains the value with its string
NL	Field does not contain the value with its string
LT	Field is less than to value
GT	Field is greater than the value
LE	Field is less than or equal to value
GE	Field is greater or equal than the value

Clauses may be joined by AND, OR and NOT (i.e. name EQ John AND age LT 20 OR sex EQ male). Consider the following example from a prior XML cookbook recipe:

```
name                sex          age
bob                 male         22
ted                 male         43
carol               female       33
alice               female       23
```

This query would place “bob” and “ted” in the list called $myList., ordered by their ages:

```
<resource id=”myData” type=”xml” src=”data/1234-my.xml”/>
query($myList,myData,name,sex EQ male,age)
status($myList)
```

RESULT-> bob,ted

This query would place “bob” and “carol” and “alice” in the list called $myList:

```
<resource id=”myData” type=”xml” src=”data/1234-my.xml”/>
query($myList,myData,name,age LT 40,age)
status($myList)
```

RESULT -> bob,alice,carol

This query would place “alice” in the list called $myList., ordered by their ages:

```
<resource id=”myData” type=”xml” src=”data/1234-my.xml”/>
query($myList,myData,name,age LT 40 AND sex NE male,age)
status($myList)
```

RESULT -> alice

This query would place “carol” and “bob” in the list called $myList., because they both have an o:

```
<resource id=”myData” type=”xml” src=”data/1234-my.xml”/>
query($myList,myData,name,name LK o,age)
status($myList)
```

RESULT -> carol,bob

15. How to make a Control Panel

A control panel consists of 4 basic items. The **controlpanel**, the **frame** that set’s its size and position, a **textformat** that dictates the text, and finally a series of **items** that form the controls. The **controlpanel** has a *title*, does it start out *open*, and is it *closable*. The frame sets the size and colors, as well as if the controlpanel will *dock* on a side. Make sure you make the frame large enough for the number of items you will add. The **textformat** sets the text, and particularly the leading (inter-item spacing).

The workers in a control box are the **item**s. The **type** choices are buton (a button), buttonbar, checkbox, color (a color selector), combobox, header, legend, line, query, radio, search, slider, or text. Each **item** has a *title* field associated with it that can be made *bold*. The *def* can be set to the desired startup state, such as a checkbox or radio button. Typically, you call some GLUE script using the glue attribute. This script will be called each time the screen refreshes.

```
<controlpanel title="Controls">
        <frame alpha="75" frameCol="0x999999" docking="right" wid="140" hgt="240" />
        <item glue="showPic" type="checkbox" title="Show Pirates" />
        <item glue="showInfoBox" type="checkbox" title="Show Info Box" />
        <item glue="showGraph" type="checkbox" title="Show Graph" />
</controlpanel>
```

16. How to Make Graphs

17. How to Make Concept Maps

First, add "cmap" as a new item to "View." Then add an "id" to the Cmap as its title. This "id" will allow the glue you add later to refer back to this cmap. For this example, the "id" is "mccall" to identify the concept map as McCall's relationship map. To block out the background of the base map when you pull up the cmap, add "backCol" from the cmap attributes list and select a cream color. The attributes "cx" and "cy" set the position for the center dot of the cmap. To create a radial map, add a "shape" attribute to the cmap and make the value "radial." The "wid" attribute determines the size of the cmap within the browser's frame.

```
<cmap wid="350" shape="radial" cy="250" cx="400" backCol="0xFFFFCC" id="mccall">
        <dot lab="Archibald McCall" style="icon:person" id="dot0" />
        <dot lab="Tench Coxe" glue="showMe?Tench" id="dot1" />
        <dot lab="War Department" glue="showMe?War" id="dot2" />
        <dot lab="U.S. glue="showMe?Navy" style="icon:person" id="dot3" />
        <line from="" to="dot0" style="line1" />
        <line from="dot0" to="dot1" />
        <line from="dot0" to="dot2" />
        <line from="dot0" to="dot3" />
        <linestyle wid="4" col="0x0099FF" id="line1" />
</cmap>
<glue from="mccall" id="showPhillymap">status(here)</glue>
```

To call up the cmap when a user clicks on a dot on the basemap, first add a "glue" attribute to the dot. For this example, we chose a dot for Philadelphia and labeled this glue "showPhillyMap." That means when you click on the Philadelphia dot, the browser will do whatever the glue "showPhillymap" is programmed for. Next, add a "glue" item to the "View" and, using the "id" attribute, give it the same name you assigned to the dot glue (i.e. showPhillyMap). To connect this glue to the cmap, add a "from" attribute and enter the cmap "id" as the value. Now when you click on the specified dot, the cmap will pop up.

18. Adding Maps and Vector Graphics
19. Making a Dock
20. Adding Movies

21. Changing a View via GLUE

Views can also be invisible and not associated with any particular tab. By setting the *visible* attribute to “true” and giving it an *id*, you can use GLUE to cause a view to show within the currently active tab’s screen space. If the view is a visible one, the view’s tab will be activated.

Assume you had a project that looked something like this:

```
<project>
        <view title=”This is View 1” id=”myView1” />
        <view id=”myView2” visible=”false” />
        <view title=”This is View 2” id=”myView3” />
        <view id=”myView4” visible=”false” />
        <view title=”This is View 3” id=”myView5” />
</project>

<glue id=”show4”>
        setview(myView4)
</glue>
```

There are 5 views, but because the default value of the *visible* is “true” only 3 are visible (myView1, myView3, and myView5). If we called the glue called “show4” from a click or control panel, the contents of the currently active tab’s screen would be replaced with whatever we had in the view called “myView4”.

22. Floating Concept Map Dots

Dots in a concept map item (cmap) are typically arranged automatically, but you can arbitrarily place a dot anywhere on the screen by setting the *x* and *y* **dot** attributes to a position and setting the **line**’s *dir* attribute to float. If you have specified a line style, the line will be drawn from the center of dot specified in the **line**’s *from* attribute to the center of the dot.

As an example, here is the tobacco map from the JT2 project. I added a new floating dot (dot13) that hangs off of April (dot4):

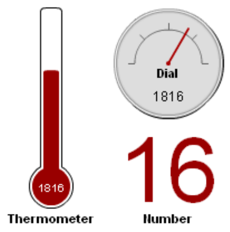
```
<cmap id="tobacMap" shape="radial" wid="420"  hgt="370"  >
   <dot id="dot0" style="leaf.gif" />
   <dot id="dot1" style="jan.gif" lab="January" />
   <dot id="dot2" style="feb.gif" lab="February" />
   <dot id="dot3" style="mar.gif" lab="March" />
   <dot id="dot4" style="apr.gif" lab="April" />
   <dot id="dot5" style="may.gif" lab="May"/>
   <dot id="dot6" style="jun.gif" lab="June" />
   <dot id="dot7" style="jul.gif" lab="July" />
   <dot id="dot8" style="aug.gif" lab="August" />
   <dot id="dot9" style="sep.gif" lab="September" />
   <dot id="dot10" style="oct.gif" lab="October" />
   <dot id="dot11" style="nov.gif" lab="November" />
   <dot id="dot12" style="dec.gif" lab="December" />
   <dot id="dot13" style="icon:letter" x="700" y="300" />
   <lineStyle id="line1" col="0x006600" type="partof" wid="3" alpha="40"/>
   <line from="" to="dot0" />
   <line style="line1" from="dot0" to="dot1" />
   <line from="dot0" to="dot2" />
   <line from="dot0" to="dot3" />
   <line from="dot0" to="dot4" />
   <line from="dot0" to="dot5" />
   <line from="dot0" to="dot6" />
   <line from="dot0" to="dot7" />
   <line from="dot0" to="dot8" />
   <line from="dot0" to="dot9" />
   <line from="dot0" to="dot10" />
   <line from="dot0" to="dot11" />
   <line from="dot0" to="dot12" />
   <line from="dot4" to="dot13" dir="float" />
  </cmap>
```

23. Adding Widgets

Widgets are a type of graph that graphically displays a single continuous value on the screen, such as a dial, clock, thermometer, etc. The range of widgets available will grow with time, but they plot the *val* attribute from *min* to *max.* The data is plotted in the color *col*. Here is the script for the following:

```
        <resource


              left="75"
        max="1822"
        <glue
        <resource
title="Dial" left="140"
               wid="100"
        <glue
        <resource
        title="Number"
               top="320" hgt="80" wid="100"/>
        <glue from="myNumber" init="true">
               sub($num,$$curYear,1800)
               set(*myNumber.dataVal,$num)
        </glue>
```



```
id="myThemo" type="widget"
style="thermometer" title="Thermometer"
top="220" wid="20" hgt="160" min="1804"
val=”1804”/>
from="myThemo" init="true" />
id="myDial" type="widget" style"dial"
top="220"
min="1804" max="1822" val=”1804”/>
from="myDial" init="true" />
id="myNumber" type="widget" style="number"
left="140"
```

24. Animating Image Views

Crops are a form of Widget that gives you a panel to display an image within. That image can be dynamically panned and zoomed much like a Ken-burns documentary. The frame item sets size of the display and you use the move() glue method to move an image within that frame.

The following script creates a crop widget with an image called “myPic.jpg”, defines a starting position and size using *left*, *top* and *wid*. Note that these are in terms of pixels in the original image.

The glue method moves the image with that frame from 3300,656 to1650,1680 and keeping the width a constant 400 pixels. The current spot in the timeline animates between those positions by using the global list called $$now, which goes from 0 to 1 as the timeline moves from left to right. The final “3” tells move to “cushion” the move to start up and slow down gracefully.

```
<resource type="widget" style="crop" id="myCrop" src="myPic.jpg" left="3300"
    top="656" wid="400”>
       <frame wid="500" hgt="300" left="12" top="28" corner="8" frameWid="2" />
</resource>
<glue from="myCrop" init="true">
       move(myCrop,3300,656,400,1650,1680,400,$$now,3)
</glue>
```

24. Show Play Button without Timeline

If you want to put a player button up without the timeline attached, to animate things, set the *hgt* and *wid* attributes of its **frame** to “0”

```
<timeline>
       <frame backCol="0x999999" top="100” left="125" wid="0" hgt="0" />
                                   </timeline>
```

Appendix

FORMATTING INFOBOXs and BOOKLETs

The text displayed in the InfoBox and Document Reader can be easily controlled using a subset of HTML tags shown below. Tables can be made by using <tab>'s to line up the. In the text you can use the following mark-up tags to control how the text will look:

- **b(text)** will bold the text within the parentheses
- **i(text)** will italicize the text within the parentheses
- **u(text)** will underline the text within the parentheses
- **font(face,size,color)** will size and color the text that follows
- **sp(leading,indent,tapstops)** will set the leading, indent & tabstops for text that follows
- **t()** will add a tab
- **br()** will add a line break
- **link(url,text)** will add a link to url when the text in the parentheses is clicked. If the url starts with *http://,* a new browser window will open and display the page, otherwise, VisualEyes assumes it is the name of a <glue> item and calls that <glue> item.
- **align(side)** will set the alignment for the text that follows. Can be *left, right* or *center*
- **img(url)** will show the image at the specified url

SPECIAL CHARACTERS

&lt;	**<** (less than)
&gt;	**>** (greater than)
&amp;	**&** (ampersand)
&quot;	**"** (double quotes)
&apos;	**'** (apostrophe, single quote)

TAGS

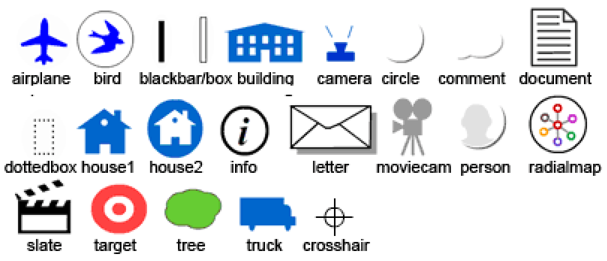
Whenever possible, use the macros above to format your text, but you can also use the following tags:

Anchor:	<a href="/support/flash/ts/documents/url">
Bold:	<b>
Font:	< font [color="#xxxxxx"] [face="Type Face"] [size="Type Size"]>
Italic:	<I>
Paragraph:	<p [align="left" | "right" |"center"]>
Underline:	<u>
Break:	<br>
Image:	<img src="/images/flash/dogs.jpg
List Item:	<li>
Tab	<tab>
TextFormat	The <textformat> tag has the following attributes:

- **blockindent** Specifies the block indentation in points.
- **indent** Specifies the indentation from the left margin to the first.
- **leading** Specifies the amount of leading (vertical space) between lines.
- **leftmargin** Specifies the left margin of the paragraph, in points.
- **rightmargin** Specifies the right margin of the paragraph, in points.
- **tabstops** Specifies custom tab stops as an array of non-negative integers.

Icon Types

There are a number of icons that can be used on dots and markers. By adding *icon:* to the name (i.e. *icon:comment*) they can be used as vector-based artwork. They can be scaled up and down by setting the *wid=""* tag to the desired width, and rotated any using the *rot=""* tag. If you don't specify a size, the width listed will be used. You can color icons using the *icol* attribute in a **<dot>**.



Name	Description	Width
airplane	A blue airplane (top view**)**	90
bird	A blue bird flying (top view**)**	90
blackbar	A filled black bar	10
blackbox	A hollow black box	10
building	A blue office building	90
circle	A white circle with grey drop-shadow	95
crosshair	A black circle with a crosshair in it	40
comment	A white circle with "tail" and grey drop-shadow	86
document	A white box with page lines and "dog-eared" corner	20
dottedbox	A hollow black box drawn with dot	20
house1	A blue house	35
house2	A white house in a blue circle	35
house3	A white house in a red circle	35
info	A white circle with "I" and black border	30
letter	A white envelope with black lines and grey drop-shadow	54
moviecam	A grey drawing of a movie camera	53
person	A white circle with grey drop-shadow and grey silhouette	95
radialmap	Hub and spokes in white circle	50
slate	Black and white movie slate	30
tree	Architectural Green tree (plan view)	20
truck	A blue moving van	80

Dot Style Types

There are a number of drawn shapes that can be used on dots and markers. They can be scaled up and down by setting the *wid=""* tag to the desired width. Setting the *col=""* tag will set the color they will be drawn in.

Name	Description
bar	A filled bar
but	A filled bar with circular ends
cir	A filled circle
rbar	A filled bar with rounded corners
star	A filled 5-point star
trid	A filled triangle facing down
tril	A filled triangle facing left
trir	A filled triangle facing right
triu	A filled triangle facing up

XML Data Format

The easiest way to import data into VisualEyes is using Excel to create a spreadsheet. The top line should contain the field names and the following lines that data for those fields. For example, this format defines 3 fields, name, sex, age and has 4 people's information:

```
name                sex                 age
bob                 male                22
ted                 male                43
carol               female      33
alice               female      23
```

Save this out as a tab-delimited text file in Excel by selecting "Save As…" in the File menu and setting the "Save as type" option to "Text- (Tab delimitated)" and saving to a file. The VisEdit editor has a tool that will allow you to load that file from your computer to the screen area, where it will be formatted into an XML format like this:

```
<TABLE a="name" b="sex" c="age">
      <ROW a="bob" b="male" c="22" />
      <ROW a="ted" b="male" c="43" />
      <ROW a="carol" b="female" c="33" />
      <ROW a="alice" b="female" c="23" />
</TABLE>
```

Click on the "Upload to server" button and that file will be saved on VisualEyes server's data folder using your user id and a name you gave it (i.e*. /data/1234-MyXMLFile.xml*).

Web Table Data Import

Existing websites are a great good of data for projects, for example the Historical Census Browser (http://fisher.lib.virginia.edu/collections/stats/histcensus) is a great way to get county-level census data from 1790 to 1960. Once you have found a table of data you want, select the entire table and copy (CTRL-C) it into your computer's clipboard.

Paste (CTRL-V) this data in an open Excel spreadsheet. The first line should contain a list of single-word field names that each column can be referred by (i.e. name, sex, age in the precious example).

Save this out as a CSV (comma delimited values) or tab-delimited text file in Excel by selecting "Save As…" in the File menu and setting the "Save as type" option to CVS (Comma delimited) or "Text- (Tab delimited)" and saving to a file.

In the Tools section of the VisEdit editor, select the *Convert Data File to XML* option. Make whatever change you need to the raw text. Click the *Convert to XML* button. Edit the field names on the first line so that they do not contain any spaces. Click on the "Upload to server" button and that file will be saved on VisualEyes server's data folder using your user id and a name you gave it (i.e*. http://www.viseyes.org/data/1234-MyXMLFile.xml*).

Embedding VisualEyes Projects in Web-pages

The Easy Way - While you can always show your VisualEyes project using our website using the www.viseyes.org/show?id=xxxx link, you can also embed it more seamlessly into your own website by adding 2 lines to the page you want to embed the project on your site:

```
<script> var id="xxxx";   var bcol="#ffffff";   var wmode="opaque"; </script>
<script src="http://www.viseyes.org/embed.js"></script>
```

Replace the ***xxxx*** with your actual project number, which is show at the top right corner of the VisEdit screen. If you want a different background color than white, replace the ***ffffff*** with the color you want behind the tabs and under the timeline. You can make that area transparent by changing **opaque** to **transparent**.

The Hard Way - If you want to put a copy of the actual SWF file on your page, the process is more complicated. The reason to do this is to freeze the version of VisualEyes and ensure the version you are embedding will not change. This is important for museums and other institutions that require the utmost of stability.

To do this, you will need to add two files from us, the SWF file called VisualEyes.swf and a JavaScript file called “localembed.js.” Email me at *bferster@virginia.edu* for them.

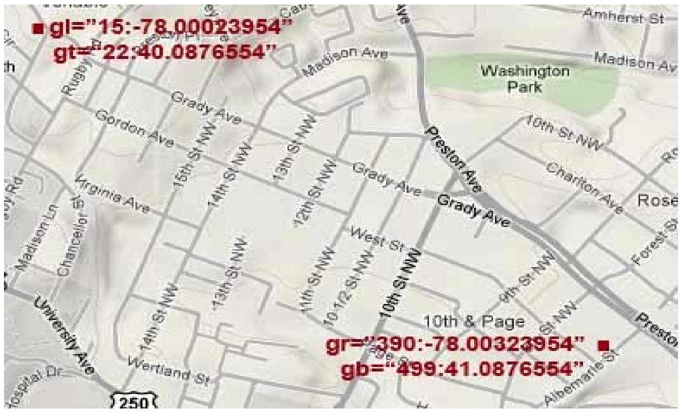
Put both files in the same folder as your webpage you want to embed the project in and add the following lines to that file:

```
<script> var id="xxxx";   var bcol="#ffffff";   var wmode="opaque"; </script>
<script src="localembed.js"></script>
```

Replace the ***xxxx*** with your actual project number, which is show at the top right corner of the VisEdit screen. If you want a different background color than white, replace the ***ffffff*** with the color you want behind the tabs and under the timeline. You can make that area transparent by changing **opaque** to **transparent**.

Geo-Rectifying Maps and Images

You can specify dot coordinates in longitude and latitude, instead of specifying the coordinates in x and y pixel values.



This makes it easier to use locations from Google Maps and other GIS-aware applications.

To do this, we need geo-rectify (a GIS term) the map that the dots will be placed over to correlate the longitude and latitude values to their position on the map or image.

1. Choose two points on the map, one in the upper left corner, and one in the bottom right.

2. Get the pixel position for each point by clicking on the point and pressing the "Alt" key when viewing the project. The screen position will appear in the bottom right of the screen.

3. Find the latitude and longitude and for each point. The longitude (a negative number for US locations)

4. Add 4 new attributes to the image or map item, matching a pixel value to a geo coordinate, separated by a colon. The *gl* is the left side, *gr* the right side, *gt* the top, and *gb* the bottom **(i.e. gl="49:-78.500488" gt="41:38.041771" gr="759:-78.46872" gb="460:38.027124")**

5. You can now specify the dots x and y attribute in longitude and latitude coordinates (i.e. x="-78.500488" y="38.027124"). In the **path** item containing dots add an attribute called res to tell the path to rectify the dot to the particular resource you added the gl/gt/gb/gr attributes to (i.e. res="myMap).

Here is an example for the eastern US. The top left corner was set at Columbus, OH, and the bottom right corner at Virginia Beach, VA, with dots in Charlottesville and NYC:

```
<resource gb="545:36.84446074079564" gr="751:-76.00341796875"
    gt="186:39.9434364619742" gl="112:-83.0126953125"
    src="http://farm5.static.flickr.com/4007/4334889804_bb1edf385f_o.jpg" id="baseMap"
    type="image" />
<glue init="true" from="baseMap" />
<path res="baseMap" id="myPath">
    <dot col="0x990000" wid="10" style="cir" y="39.9434364619742" x="-83.0126953125"/>
    <dot y="36.84446074079564" x="-76.00341796875" />
    <dot col="0x00CC00" y="40.713955826286046" x="-74.02587890625" />
    <dot col="0x00CC00" y="38.02689818226723" x="-78.48585605621338" />
</path>
<glue from="myPath" init="true" />
```

Understanding XML

XML (eXtensible Markup Language) has become very popular as a way to request, retrieve, and show data in a webpage. Because it is just plain text, people can write XML in almost any word processor or text editor. It is not a language, but more of a storage format for information.

XML uses a tagged format, and resembles HTML in its formatting and structure, but instead of a series of specific tags such a <b> to bold text, the tags are created by the person writing the XML to put the information into "slots", much like sorting the mail. The act of tagging the information into a format gives it structure so a computer can read the document and know what is what.

Elements and Attributes

An XML document is made up of a number of ***Elements*** that contain the information. Each element has a tagged name and begins with an opening tag enclosed in angle brackets, like this <album>, and ends in a closing tag like this </album>. Inside the opening tag, the node can contain ***Attributes*** that define the properties of the information you want to store. For example, an album might contain attributes such as the title, the artist, and a year, for example:

```
<album title="Yer Blues" artist="Glen Bull" date="2009">
</album>
```

In this example, album is the name of the element; title, artist, and year are attributes. The information following the equals sign must be put in quotes. Attribute and element names cannot contain spaces, but people typically join the words together capitalizing the middle ones like this: aVeryGoodAlbum, called ***Camel Case,*** because the capitals represented the humps on a camel to someone. In between the opening and closing can be other elements, or content. If there is no content or notes between, the opening closing can be joined together.

```
<album title="Yer Blues" artist="Glen Bull" date="2009">
    <song name="I Got the Blues"/>
</album>
```

Text

We can add some ***text*** to the node by encasing it in a <!CDATA[[...]]> wrapper tag, which allows you to write any kind of characters within the brackets without worrying about disrupting the XML formatting.

```
<album title="Yer Blues" artist="Glen Bull" date="2009">
    <song name="I Got the Blues"/>
    <song name="Gina on my Mind"/>
    <!CDATA[[This album was the was Bull's comeback]]>
</album>
```

There really isn't that much more to XML than the simple ideas of ***elements***, ***attributes*** and ***text***, but with these three ideas, you can describe very complicated relationships between things. The parent-child metaphor is often used to describe the relationship between elements and attributes. In the example above we have an album (the ***parent***) that contains two songs (the ***children***) because the nodes for the songs are nested within the album. This kind of nesting can go on and on ad infinitum to simply represent very complex relationships between elements that are obvious to the naked eye and to the computer.

Some Fine Points about XML

1. Most XML documents begin with an element like this: <?xml version="1.0" encoding="utf-8" ?> that tell the encoding and version of XML you are using.

2. You can add comments to the document that help describe what is going on like this: <!-- This is a comment -->.

3. XML documents can only contain one top level element. If <album> is the top-most element, having two of them would make the document invalid, however having two sub-elements with the same name, such as <song> is fine. If you wanted to make a document with multiple albums, you would have to add an element to hold them all, like this:

```
<?xml version="1.0" encoding="utf-8" ?>
   <albumSet>
      <album title="Yer Blues" artist="Glen Bull" date="2009">
         <song name="I got the Blues"/>
      </album>
      <album title="Teacher, Teacher" artist="Glen Bull" date="2001">
        <song name="What Ex-CITE-ment!"/>
      </album>
      <!-- This is comment -->
</albumSet>
```

4. XML is ***very*** picky about its format and does not tolerate from missing angle brackets or quote marks.

5. Some word processors like Microsoft Word will occasionally put in angle quote marks that XML does not recognize as straight up quote marks.

6. Putting quotes or apostrophes within an attribute can also confuse the formatting, so they need to be preceded by a backslash, like this title="Yer\'s Blues is \"cool\"".

Writing and Checking XML

If you have access to Adobe Dreamweaver or any other text editor, it's good to XML layer because the editors have color coding that highlights correct formatting. There is an checker for validating the formatting of your XML on the Web: http://www.w3schools.com/XML/xml_validator.asp. Cut-and-paste your XML into the area marked "Syntax-Check Your XML" and any errors will be displayed.

Using VisEdit

Sharing Your Project with Others

The adding the project number to the following URL will allow anyone with an Internet connected web browser that has the Flash plug-in installed to see your project:

www.viseyes.org/show?id=xxxx

Replace the *xxxx* with your actual project number, which is show at the top right corner of the VisEdit screen.

Moving Items in the Main Tree View

The **UpArrow** and **DnArrow** keys will allow you to navigate through the Main Tree view of your project. The Home key will bring you to the top. To make editing the XML directly less needed, using the **Shift-UpArrow** or **Shift-DnArrow** keys will move the currently selected item up or down in the Tree. Any sub-items within that item will also be moved. There are options in the Edit menu for doing this as well.

Copy and Paste of Items in Main Tree View

To make editing the XML directly less needed, selecting the Copy option from the Edit menu (or hitting **Ctrl-C**) will copy currently the selected item to the clipboard. Any sub-items within that item will also be copied. Selecting the Paste option from the Edit menu (or hitting **Ctrl-V**) will copy any items in the clipboard just beneath currently the selected item in the Tree.

Undo/Redo in Main Tree View

You can click on the **Undo** option in the **Edit** menu to go back to a previous step. This works similarly as in programs like Microsoft Word. You can go back to your last 100 actions. Selecting the **Redo** item in the **Edit** menu will "undo" the undo.

You can also undo the last 32 keyboard actions by hitting the **Ctrl-Z** key. This key is also available when editing **<glue>** and **<infoBox>** script areas.

Data Import from Many-Eyes

IBM's Visual Communication Lab has a great free website (www.many-eyes.com) for visualizing, storing and sharing data sets. You can automatically pull them in as an xml data source by using a link to its data file. You can import these data sets dynamically into History browser by locating a data set or uploading your own to their site. Click on the link called Data File and use that URL when defining an xml resource. This is a simple example of a data set on ManyEyes:

```
<resource type="xml" id=myData" src="http://manyeyes.alphaworks.ibm.com/manyeyes/datasets/things-2/versions/1"/>
```

Alternatively, you can copy the data on our server by doing the following: Add ".txt" to the url in a web browser, (ie http://manyeyes.alphaworks.ibm.com/manyeyes/datasets/things-2/versions/1.txt) and highlight the data manually (CTRL-A) and copy the data (CTRL-C) onto your computer's clipboard.

In the Tools section of VisEdit, select the *Convert Data File to XML* option. Instead of loading a file to convert, click “Cancel” and paste (CTRL-V) the data over the instructions in the text box. Click the *Convert to XML* button. Edit the field names on the first line so that they do not contain any spaces. Click on the “Upload to server” button and that file will be saved on VisualEyes server’s data folder using your user id and a name you gave it (i.e. *http://www.viseyes.org/data/1234-MyXMLFile.xml*).

Upload XML Project File Directly

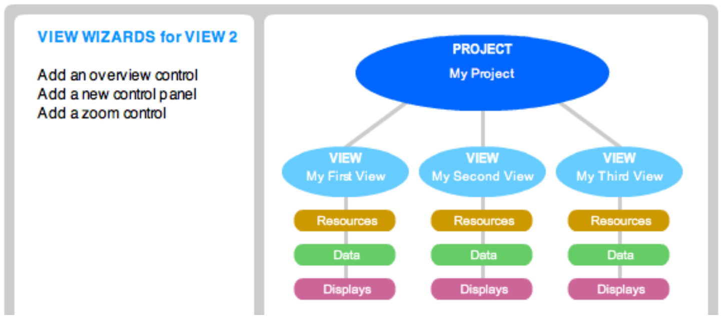
While it is possible to edit the XML directly using the VisEdit editor, many people making projects will feel more comfortable editing the XML in a text editor such Oxygen or DreamWeaver. These editors have good undo/redo and context coloring the make the process much easier.

To support this work flow, there is an option in the VisEdit File menu called “Upload Local XML File” which will bring up a file box and allow you to select an XML file from your computer’s hard drive and upload it to your currently active project. Once uploaded, it will open the same browser window that the “Save and Preview” button uses to preview the project.

The flow goes like this: 1) Edit XML in DreamWeaver 2) Save file to disk in DW 3) Upload Local XML File in VisEdit editor 4) See how it looks 5) Go back to step 1.

Using Wizards

Wizards are step-by-step guides that help the adding of sections of XML to projects. Clicking on the “Wizard” option in the “Show” menu will bring up the wizard home page. This home page is where you choose the various tasks we have made available as step-by step-wizards.



The large portion of the screen contains a tree map of the views currently in your project. Clicking on the *Project* bubble lists the wizards available to the project as a whole in the sidebar to the left.

Clicking on any of the *View* bubbles and below will list the wizards available to add to views.

Clicking on a wizard in the sidebar will walk you through its steps, prompting you to type something, pick an option, or a color. As you go, the XML is built in the lower panel. When done, click on the “Add script” option to add it or the “Quit with out adding option. Use the arrow keys, the “Previous Step” and “Next Step”, or clicking on the steps directly to navigate through the steps.