



GLUE

9/6/10

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: To cause resources, such as images, paths, and charts, to show up on the screen, automatically or on command and 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 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 is drawn.
4. *Once* causes the **GLUE** run only once (useful for initialization).

Aside from the 4 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, 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 a **controlPanel** item, the *id* can be 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 many 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 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

VisualEyes, all of the activities or methods except for the `list()` method expect a 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 *years* you will want to use and to which you will want to associate your data. Hence, the *year* **varies** because it's a **variable**. 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



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.

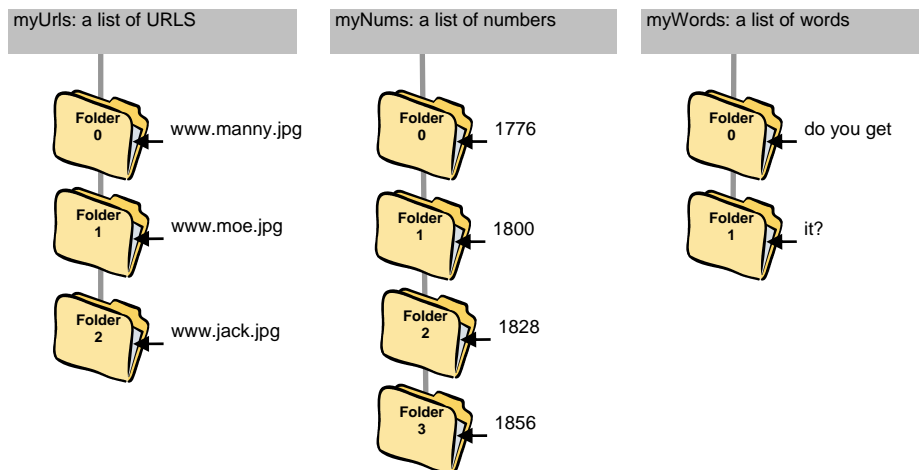
Lists are named containers that hold many kinds and number of items



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.

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.

For example, here are three lists:



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 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
\$\$zoomBox	Coord's of box chosen in by zoom magnifier (left,top,right,bot)

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
```

Special Characters in glue scripts

When you want to use HTML macros within a glue script, you have to be careful about two characters interfering with the parsing of the script, namely the comma and the right parenthesis. If you need to put a `)` in a script, use the `~` (tilde) instead, and use ```` (accent) in place of a `,`.

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).

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: a number, word(s), 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 met -- 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.

These 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 (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. Putting a *minus sign* before the field (i.e. "-age") will sort the rows in descending order.

6. The results

The results are a list of items that your query found. This is usually in the form of a *list* object, but you can also provide a field in another table to capture the results by providing the table's name and the field within it you want to fill with the results of the query. Add a star, the table name a dot and the field name like this: **myTable.age* would fill the *age* field in the *myTable* table with the results from the query.

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.

```
<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

GLUE METHODS REFERENCE

The following GLUE methods are available. They are functionally listed below, followed by an alphabetical list. Be sure and include ALL parameters shown in the documentation.

List Management

copy
list
listfill
listjoin
listmerge
listnum
listsplit
lookup
segment
select
set
tweenlist

String Management

datetodays
daystodata
join
replace
replaceword
split

Math

abs
add
div
inc
floor
max
min
mul
random
round
sqrt
sub

Statistics

average
stdev
correlate

Logic

call
if
repeat
setview

Data Management

dataset
datetime
dotfill
filldocviewer
linefill
normalizograph
query
routefill
table

Everything else

dissolve
gototime
link
menuitem
move
movie
play
radioshow
refresh
setimage
show
status
tween



abs(dest, source)

This method takes the absolute value (ie. positive numbers only) of source and places the result in the list called *dest*. If *dest* list does not exist, it is created.

dest Name of list to store result
source Number to abs

abs(\$age,\$age)

average(dest, data)

This method averages numbers in *data* and places the result in the list called *dest*.

dest Name of list to store result
data List of number to average

average(\$avg,\$myList)

bandfill(path, dataRes, bandNum, [start])

This method will fill a container object, such as a path or concept with dot data from a data source (i.e. an XML file, or *table resource*). See the dot specification for more information. The *bandNum* specifies which band to load. By default, dots will be added to the dots already in the timeview or shelf, making it convenient to specify the first dot, and letting the table to be loaded only have the *dot* attributes that change, since they will be inherited from the first. Setting the start parameter to "1" will leave the first *dot* as is and fill beyond it.

path ID of timeview display
dataRes ID of resource where data are
bandNum Number of band to fill
start Dot number to start filling path at (optional)

bandfill(myTimeView,myData,0)

call(glue)

This method will call a **GLUE** method, like a subroutine.

glue ID of **GLUE** method to call

*call(my**GLUE**)*

copy(dest, source)

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

dest ID of list to or resource to copy to
source ID of list to or resource to copy from

copy(\$to,\$from)

correlate(dest, data1, data2)

This method performs a Pearson's product-moment coefficient of correlation between numbers in *data1* and *data2* and places the result in the list called *dest*.

dest Name of list to store result
data1 List of numbers in data set 1
data2 List of numbers in data set 2

correlate(\$cor,\$myList1, \$myList2)

dataset(graph, set, legend, dataRes)

This method adds a row of data to a graph. If the set number is set to clear, all the sets will be removed from the graph.

graph ID of graph
set index of dataset
legend Name of legend
dataRes ID of resource where data are

dataset(myGraph,2,Sales,\$mydata)

datetime(graph,time)

This method controls how much of a graph to show, to help make charts that grow over time. The time parameter goes from 0 to 1.

graph ID of graph
time Amount to show (0-1)

datetime(myGraph,.5)

datetodays(days, date)

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.

days ID of list to put days into
date Days

datetodays(\$days, 1/1/2009)

daystodate (date, days, format)

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

date ID of list to put date into
days Days
format Date format

daystodate(\$date, 3650, mo/yr)

dissolve(in, out, start, end, dur)

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.

in ID of incoming resource
out ID of outgoing resource
start Start time of outgoing res (0-1)
end Start time of incoming res (0-1)
dur Duration of dissolve transition (0-1)

dissolve(\$pic1, pic2, 0, .5, 1)

div(dest, var1, var2)

This method divides *var1* by *var2* and places the result in the list called *dest* (i.e. $dest=var1/var$).

dest ID of list to store result
var1 Number to be divided
var2 Number to divide

div(\$pct, \$sum, 100)

dotfill(path, data, [start])

This method will fill a container object, such as a path or concept with dot data from a data source (i.e. an XML file, or *table resource*). See the dot specification for more information. By default, dots will be added to the dots already in the path, making it convenient to specify the first dot, and letting the table to be loaded only have the *dot* attributes that change, since they will be inherited from the first. Setting the start parameter to "1" will leave the first *dot* as is and fill beyond it.

<i>path</i>	ID of path
<i>dataRes</i>	ID of resource where data are
<i>start</i>	Dot number to start filling path at (optional)

dotfill(myPath,myData)

featureid(mapID, idList)

This method will replace the *id* attribute of each of the features in a *map* resource with a list specified by the *idList* parameter.

<i>mapID</i>	ID of map resource
<i>idList</i>	List of map ids to replace

featureid(myMap,\$ids)

docfillviewer(docViewer, title, data)

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).

<i>docViewer</i>	ID of viewer
<i>title</i>	Name title page to fill
<i>dataRes</i>	ID of resource where data are

docfillviewer(myDocViewer,Title of it,myData)

floor(var)

The round method returns the rounded value of *var* and places the back in source.

<i>destID</i>	Name of list to store result
<i>var</i>	Number to be floored

floor(123.456)

gototime(days)

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

days When to go on the timeline

gototime(4000)

if(var1, condition, var2, lines)

This method will execute the number of lines specified if condition between var1 and var2 is met.

NOTE: There is no space between "if" and "("!

var1 Test 1
condition Condition (GT, LT, EQ NE, LE GE,LK,NL)
var2 Test 2

if(\$age,EQ,34,2)

inc(dest)

This method increments the number in the list called val by one. (i. e. val=val+1)

val ID of list to increment

inc(\$count)

join(dest, str1, str2, str3)

This method will join 3 strings together return the combined list in the first parameter.

dest ID of list hold combined strings
str1 1st string to combine
str2 2nd string to combine
str3 3rd string to combine

join(\$dest,\$first,\$second,A 3rd literal)

linefill(line,data)

Fill lines from a data table

line	Name of line object to fill
data	Name of data table

linefill(myLine,myData)

link(url, target, clickParam)

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`.

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). 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.

<i>url</i>	Full URL of page to load, or ID name of list
<i>target</i>	browser window or frame (<code>_self</code> , <code>_blank</code> or <code>_overlay</code>)
<i>clickParam</i>	If set to true, <code>?id=</code> will be added to url

link(www.mysite.org,_blank,false)

list(val1, val2, ... valN)

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. All lists is names are preceded by a dollar sign (i.e. `$myList`) Global lists (meaning their scope is view-wide) are preceded by 2 dollar signs (i.e. `$$myGlobalList`).

<i>listID</i>	Name of list l(<code>\$(+name=local)</code> , <code>\$(+name=global)</code>)
<i>val1</i>	Element to add to list
<i>val2</i>	Element to add to list
<i>...</i>	Any number of elements
<i>valueN</i>	Element to add to list

list(\$years,1865,1866,1877,\$id,\$\$param)

listfill(dest, source, match, default)

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

<i>dest</i>	Name of list to store result
<i>source</i>	Name of list
<i>match</i>	Value to set matching indices in <i>dest</i> to
<i>default</i>	Value to set all other indices in <i>dest</i> to

listfill(\$a,\$b,yes,no)

listjoin(dest,source)

This method will join the second list to the end of the first list and return the combined list in the first parameter.

<i>dest</i>	Name of first list to combine
<i>source</i>	Name of second list to combine

listjoin(\$first,\$second)

listmerge(dest, source, spacer)

This method will join the separated members of a list separated by any spacer set and return that string in the first parameter.

<i>dest</i>	Name of list to store combined
<i>source</i>	Name of list
<i>spacer</i>	Value to between entries

listmerge(\$joined,\$separate,)

listnum(num,source)

This method puts the length of the *source* list into the *dest* list to get the number of members it holds. If *dest* does not exist, it is created.

<i>num</i>	ID of list to place count in
<i>source</i>	ID of list to count members of

listnum(\$num,\$myList)

listsplit

Split list members by token

listsplit(source,separator)

This method will look at each member of *source* list, and if it contains the separator, the member will be split and added as a new member of the list.

<i>source</i>	ID of list	
<i>separator</i>	String	Separator character or string

listsplit(\$myList,|)

lookup

Look up value in table

lookup(dest, source, find, deliver)

This method will search for a value in a table's field, and return a different field on the same row of that table.

<i>dest</i>	Name of list to store result
<i>source</i>	Name of list or resource data
<i>find</i>	Field in data set to search in
<i>deliver</i>	Field in data set to return

lookup(\$t,Lincoln,myTable,name,age)

menuitem

Change control panel item

menuitem(control, A new title, GLUE, value)

This method will change an item in a control panel to a new title, **GLUE** or value. Setting a parameter "undefined" keeps its old value.

<i>control</i>	ID of control panel item	
<i>title</i>	String	New title
GLUE	String	New GLUE method
<i>value</i>	String Number	Value to set item to

menuitem(myCheckBox,A new title,undefined,true)

max

Find maximum of two numbers

max(dest, var1, var2)

This method compares var1 by var2 and places the largest in the list called *dest*.

<i>dest</i>	Name of list to store result
<i>var1</i>	Number to be compared
<i>var2</i>	Number to be compared

max(\$biggest,\$sum,100)

min

Find minimum of two number

min(dest, var1, var2)

This method compares var1 by var2 and places the smallest in the list called *dest*.

<i>dest</i>	Name of list to store result
<i>var1</i>	Number to be compared
<i>var2</i>	Number to be compared

min(\$smallest,\$sum,100)

move

Move a resource over time

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

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

<i>resource</i>	ID of resource or ""screen"";
<i>startX</i>	Starting horizontal position
<i>startY</i>	Starting vertical position
<i>startZ</i>	starting zoom percent
<i>endX</i>	Ending horizontal position
<i>endY</i>	Ending vertical position
<i>endZ</i>	Ending zoom percent
<i>timing</i>	ID of timing source (i.e. timeline, var, 0)
<i>eases</i>	Motion slows (0=none1=start 2=end 3=both)

move(\$myPic,100,200,100,200,300,150,\$\$param,3)

movie

Control a movie resource

movie(player, command, value)

This method will control a movie resources transport functions such as play or stop. Current movie commands are 1.) play - The param is the time in seconds to start playing the movie from. 2.) stop, 3.) seek - The param is set to the time in seconds to cue the movie to. 4.) time - The param the name of the list to store the current time in seconds. 5.) start - The param is the time in seconds of movie's start time. 6.) end - The param is the time in seconds of movies end time. 7.) load - The param is the src/path of the movie to load.

<i>player</i>	Name of movie to control
<i>command</i>	Command to send to player
<i>value</i>	Value to send to player

movie(myMovie,play,12)

mul(dest, var1, var2)

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

<i>dest</i>	Name of list to store result
<i>var1</i>	Number to multiply
<i>var2</i>	Number to multiply

mul(\$tot,\$age,10)

normalizegraph(graph, max)

This method will set the status of a graph set by graphID to plot the data as raw numbers by setting max to 0 (its 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.

<i>graph</i>	ID of graph
<i>max</i>	Maximum value of Y axis

normalizegraph(\$myGraph,100)

play(startTime)

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.

<i>startTime</i>	Starting time
------------------	---------------

play(1/1/1780)

query

Queries a data set and returns results in a list

query(result, tableRes ,fields, conditions, orderBy)

This method works in a similar fashion to a SQL query on a table, but performs conditional searches on data contained in a resource called *tableRes* that is in row/column format. The results are placed in a new list specified by *results*. You can specify the fields to include using the *fields* id. A * will include all fields. Multiple fields are set by separating them with plus signs. The conditions are similar to SQL WHERE conditions, with AND indicating "AND" and "OR" indicating OR. Possible operators are EQ, NE, GT, LT, LE, GE, LK, NL, (like/not-like).

<i>result</i>	ID of list where results are placed
<i>tableRes</i>	ID of resource where data are
<i>fields</i>	Fields to include, separated by a + sign, or * for all
<i>conditions</i>	Inclusion conditions, separated by AND or OR
<i>orderBy</i>	Field to order row results by (0=none, add - for reverse sort)

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

random

Get a random number

random(dest, min, max, integer)

This method returns into *dest* a random number between *min* and *max*. If *integer* is set to "true", no decimal places will be added.

<i>dest</i>	Name of list to store result
<i>min</i>	Minimum number
<i>max</i>	Maximum number
<i>integer</i>	1

random(\$num,0,100,true)

radioshow

Select one resource from several

radioshow(select, opacity, resources)

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* the resources list.

<i>select</i>	Which resource index to select (0-n)
<i>opacity</i>	Opacity of resource 0-100
<i>resources</i>	ID of list of resource IDs

radioshow(3,60,\$list)

refresh(resource, [param])

This method will cause the resource identified to be re-drawn. Some elements can pass a parameter to the refresh, such as time, or item to highlight. This is optional and currently used in the *shelf resource* to cause a particular *dot* within the shelf to be highlighted.

resource ID of resource
param Optional

refresh(myPic)

repeat(times)

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.

times Number of repeats or end

repeat(4)
 - or -
repeat(end)

replace(infobox, search, replace)

This method looks at some text and replaces an occurrence of search with a word or words identified by replace.

infobox ID of infobox
search Value to look for
replace Value to replace it with

replace(myBox,jump,howfar)

replaceword(infobox, words)

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.

infobox ID of infobox

words List of replacement values

replaceword(myBox,\$words)

round

Round a number up

round(source)

This method returns rounded up of *source* and places the result back in to *source*.

source Name of list to store result and the

round(123.456)

routefill

Add routes to a path via a data element

routefill(path, dataRes)

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 the start, end, and pathway fields. See the route specification for more information.

path ID of path

dataRes ID of resource where data are

routefill(myPath,myData)

segment

Sort data via filters and values

segment(dest, source, slots, values)

This method will sort data into a number of preset categories and use those as criteria to create a new list. The *slots* contains a list of numbers that sets the ranges, and *values* (which must have the same number of items as *slots*) contains the values to use. The *source* points at a list of data to compare against the slots, and the *dest* is where the converted list goes. If the *source* contains multiple items, the each item will be converted and placed in the *dest* list. For example suppose we wanted to show some text when certain dates are reached. (i.e. "1968" will yield "The Sixties".) The *timeline* changes the year, which is reflected in the *\$\$curYear* global.

1. Making a list called \$slots like this: *list(\$dates,0,1950,1963,1975)* sets up 4 date ranges: before 1950, 1950 to 1963, 1963 to 1975 and past 1975.
2. When a year is in one of those ranges, we want to display the era, so we set up an list of values like this: *list(\$eras,The War Years,Happy Days,The Sixties,Modernity)*.
3. This call: *segment(\$name,\$\$curYear,\$dates,\$eras)* will look at the current year, decide which slot it is in and put the era's name in \$name.

<i>dest</i>	ID of destination data resource
<i>source</i>	ID of source data resource
<i>slots</i>	ID of list of slots data
<i>values</i>	ID of list of values to assign segmented data

segment(myData.pop, myMap.col, \$slots, \$colors)

select Copies the nth member from source list to dest list

select(dest, source, which)

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.

<i>source</i>	ID of list of values to select fro
<i>dest</i>	String of list where selection is placed
<i>which</i>	Selection number

select(\$choices,\$dst,5)

set Copy a list element

set(dest, source)

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

<i>dest</i>	Name of list to store result
<i>source</i>	Name of source

set(\$b,\$a)

setatt This method will set an attribute of an element

setatt(element, attribute, value)

Set an element's attribute

<i>element</i>	ID of element
<i>attribute</i>	Name of attribute to set
<i>value</i>	Value of attribute to

setatt(myElement,src,\$\$param)

setdot Set a dot's attribute

setdot(path, bandNum, dotNum, field, value)

This method will set a field (attribute) of a dot in a path or TimeView band.

<i>path</i>	ID of path or timeview band
<i>bandNum</i>	Number of band (if in a TimeView)
<i>dotNum</i>	Number of dot to set

table(action, table, row, field, value)

This method will modify the contents of a *table resource*. There are three possible actions: *addrow*, which adds a row, *set*, which set's the value of an item, and *sort*, which sorts the table by a field. Row numbers start at 0.

addrow: If you put in *-1* as the *row*, it will add a new row to the end of the table, otherwise it will put it at the specified row. The *field* parameter should be a list of field values you want to add to that row.

set: The *row* parameter sets the row you want to change, *field* is the field name, and *value* is the value you want to set the item to. If you put in *-1* as the *row*, the field name in all the rows will be filled with *value*.

sort: Set *field* parameter to the field to sort by. Putting a *-1* in the value parameter will sort in descending order instead of ascending.

<i>action</i>	Table action to perform: addrow set sort
<i>table</i>	ID of table to modify
<i>row</i>	Number of row
<i>field</i>	Field(s)
<i>value</i>	Value of data (set action)

<i>table(addrow,myTable,-1,\$data,0)</i>	<i>adds new row at end</i>
<i>table(addrow,myTable,4,\$data,0)</i>	<i>adds new row at row 5</i>
<i>table(set,myTable,4,name,smith)</i>	<i>sets name field at row 5 to "smith"</i>
<i>table(sort,myTable,0,age,-1)</i>	<i>sort table by "age" field in descending order</i>

timelinelabels(dates, labels)

This method will add labels for a timeline. Add a list of dates and a list of labels to show at each date

<i>dates</i>	ID of list of dates
<i>labels</i>	ID of list of labels

timelinelabels(\$myDates,\$myLabels)

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

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.

<i>field</i>	ID of resource
<i>start</i>	Starting position

end Ending position
timing ID of timing source (i.e. timeline, var, 0)
eases Motion slows (0=none1=start 2=end 3=both)

tween(myPic.rot,100,200,\$\$param,3)

tweenlist

Animate between two lists

tweenlist(dest, from, to, percent, eases)

This method will set a list to tween between two other lists over time.

dest Name of list to store result
from Name of list to tween from"
to Name of list to tween to
percent Number Percent of tween (0-1)
eases Motion slows (0=none1=start 2=end 3=both)

tweenlist(\$myList,\$list1,\$list2,\$\$now,3)

visible

Show or hide resource

visible(resource)

This method sets the visibility of a resource. The resource can be shown (visibility=1) or hidden (visibility=0).

resource ID of resource
visibility Visibility of resource 0 or 1

visible(\$myRes,0)