A Visual Stack Based Paradigm for Visualization Environments

Matt Gilbert and Santiago V. Lombeyda

Art Center College of Design and California Institute of Technology, Pasadena, California, USA

ABSTRACT

We present a new visual paradigm for Visualization Systems, inspired by stack-based programming. Most current implementations of Visualization systems are based on directional graphs. However directional graphs as a visual representation of execution, though initially quite intuitive, quickly grow cumbersome and difficult to follow under complex examples. Our system presents the user with a simple and compact methodology of visually stacking actions directly on top of data objects as a way of creating filter scripts. We explore and address extensions to the basic paradigm to allow for: multiple data input or data output objects to and from execution action modules, execution thread jumps and loops, encapsulation, and overall execution control. We exploit the dynamic nature of current computer graphic interfaces by utilizing features such as drag-and-drop, color emphasis and object animation to indicate action, looping, message/parameter passing; to furnish an overall better understanding of the resulting laid out execution scripts.

Categories: Graphic User Interfaces, Visualization Systems, Visual Environments, Visual Programming, Object-Based Languages, Stack-Based Languages

1. Introduction

There is a growing need among researchers to find a visual interface or paradigm that offers simple visual representations to clearly encapsulate execution scripts at the same time as allowing for mounting complexity. This is true both for researchers utilizing graphic visualization systems as well as those using visual frameworks for the scripting of large simulations. We have firsthand understanding with this need at Caltech's Center for the Simulation of the Dynamic Response of Materials. Here, researchers are building a set of different simulation codes for fluid dynamics, solid mechanics, and material properties. Their goal is to couple these codes dynamically and interchangeably, create basic simulation strategies, and then visualize results.

A first attempt at a visual layer to control the integrated codes was done using IRIS Explorer to dynamically generate python scripts [Lombeyda]. However, with growing complexity, the solution was soon hindered by the basic over-simplicity and lack of flexibility of the visual paradigm and underlying system. In fact, we have further encountered similar needs throughout the research community, like the work being done as part of the Distributed Data Analysis For Neutron Scattering Experiment group at Caltech, who currently is utilizing Scripps' Viper for similar purposes.

For the most part, visual frameworks have in fact proven useful, efficient, and valuable when scripting a series of basic action nodes, modules, which visually symbolize short but complex tasks. This is due to the fact that visual paradigms aid in the understanding of how action modules interact with each other, and clearly show the expected sequence of execution. Visual paradigms are a great methodology to allow new novice users as well as experienced scientists to create flexible and dynamic pipelines of execution.

In practice, for novice users, visual environments are quite valuable as they make it easier to focus on the desired execution results without knowing the intricacies of the working of individual atomic actions. Visual frameworks empower the new user to explore, create, and test the available tools, even with minimal knowledge of the particulars.

As users grow in experience, familiarity helps them become more productive under these same environments. However, eventually when the execution models scale in complexity most environments based on the direction graph paradigm start becoming unwieldy.

Thus, our goal was to find a visual solution that would:

a) Empower both novice and expert users to create complex execution models easily

b) Bestow a better sense of ownership and control over the state of the visual system

c) Utilize the dynamic medium in a more efficient manner

The work presented here is done as part of the Visual Programming Environments (VPE) effort at Caltech. The goal of the project is to create a lightweight, purely visual framework to enable visual interfaces that can be connected to an underlying execution engine, or can in turn generate scripts (such as in Python or Perl scripts). The visual interface itself depends only on OpenGL and FreeType, while the basic event processing is done by any attached windowing system (such as GTK or GLUT), where it is only required that basic events be passed into to a central event delivery system. The visual paradigm described in this paper represents the product of the exploration for new paradigms for visual frameworks, to be then realized utilizing the previously explained VPE's framework and API.

2. Previous Work

The directional graph paradigm is commonly utilized throughout visualization systems, as it offers an intuitive way of visually displaying how outputs of a process or action module are passed to the next action module in an execution thread. Current systems that use this paradigm include IRIS Explorer, AVS, Viper, and OpenDX. However, the actual spatial placement of these modules does not carry any global additional significance, which not only fails to take advantage of the available display real estate, but once the execution model becomes increasingly cumbersome, its visual representation becomes quite crowded, and hard to understand. But, for the most part, users tend to manually arrange the layout of their visual scripts according with the directionality of the execution flow. This directionality is either in a left-to-right manner (analogous to reading) as in IRIS Explorer, or top-to-bottom (analogous to a gravity based trickling effect) as in Viper or OpenDX.

Regardless, directional graphs are in fact ubiquitous to most workflow representations, from business diagramming such as Visio, to visual simulation and modeling environments such as HyPerformix Workbench, as well as countless other examples.

But not all visualization systems use directional graphs. Paraview for instance treats each action as a separate filter, which can take its input from any other action module previously defined. These actions are shown using a simple historical list. An alternate view allows a user to view all the filters (action modules) from which data is received, and all action modules to which data is sent in a more directional graph style. In a similar fashion, CEI's EnSight, takes an object centered approach. Data sets are listed as object instances on a list of available objects/parts, on which actions can be performed. Any time a new action is performed, a new object is created, and listed.

The need for better tools led to the work of [Hils] for instance, extending directional graph flow to allow for more complex execution threads, creating a "complete" visual programming language, through the use of procedural abstraction and a specific model of execution where action flows only across "consistent boxes". But, these needs have in fact led to the exploration of radically different paradigms. As an example [Nuñez] presents a spread sheet based paradigm, where execution modules are presented as entries on a spreadsheet table, while the communication or parameter passing takes place as references to other cells. Though this solution creates tighter placement of execution modules, it complicates the users' ability to actually understand the global execution model. In fact, this solution eventually relies on an alternative directional graph view to allow the user to actually follow the laid out cell dependency threads. [Jankun-Kelly] refined the idea of spreadsheets for visualization to a solution that better suits the spreadsheet environment, by having columns represent sets of different data models, with each following column representing the next filtering (execution) stage along a visualization pipeline.

So in the pursuit of a more applicable solution to complex frameworks, we draw inspiration from two sources. First is *object-based programming*, where the methodology revolves around atomic objects upon which operations are carried out [Wegner]. In fact, this methodology encourages the breakup of programming into autonomous, easier-to-understand components. It is the intuitiveness of object-based languages that allows them to translate well to visual environments, such as modern WIMP file managers. Yet, object-based programming is powerful enough that it has led to research of its application in distributed systems. It is the basic principles found as part of this work that we base our visual solutions on; even though designing an object-based language for a distributed environment in of itself is a complex and difficult task (for which the best solution may ultimately be task dependent [Chin]).

But more prominently, we substantiate our work on the basic idea of stack-based programming languages. Stack-based languages are generally characterized by their use of stacks implicitly accessed by most operations, and by their use of postfix notation. By definition, inputs needed for an action are pushed onto the topmost of the internal stack, so when the operation is called, it simply 'pops' the elements at the top, performs the operation, and pushes the result back onto the stack. We extend the concept of stack-based languages to include assembly languages with a simple linear traversal of programming instructions with conditionals and execution jumps, and several internal stacks holding the state of the system. We explicitly denote this inclusion even though most modern CPU's implement versions of these low level languages, yet they are considered to be "raw code" and not thought of as full (high-level) programming languages. In practice, there usually exists a readable text based language with a direct one to one translation to the binary instruction. It is in this form, that we claim assembly languages as instantiations of stack-based languages.

3. Contribution

Figure A-2 depicts a real-world execution layout. On the top is the execution model as done using NAG's IRIS Explorer. On the right is the same execution pipeline done under our stacks based environment. While at first glance both representations seem overly-complex, after some inspection the stacks representation clearly exposes the task

of each individual stack, while the directional graph still remains quite inaccessible. Once each of the subtasks are well understood, following the links reveals the overall task accomplished with the execution model as a whole. This is not as easily accomplished on the directional graph representation.

4. The "Stacks"

We propose a novel visual paradigm for visualization systems, which is a marriage between stack-based programming and object-based programming systems. This paradigm is foremost intended for use for a visualization system. However, it easily allows for a more general visual programming framework. We present the basic concepts behind our stacks paradigm, while showing examples relevant to scientific visualization and image manipulation. Following we describe the basic rules of flow under the proposed system. Section 5 presents in more detail a sample working implementation done using the Processing visual language.

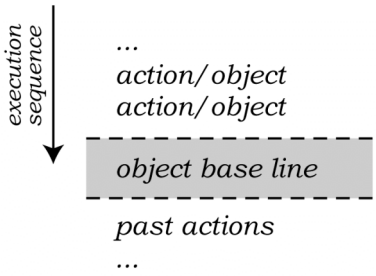
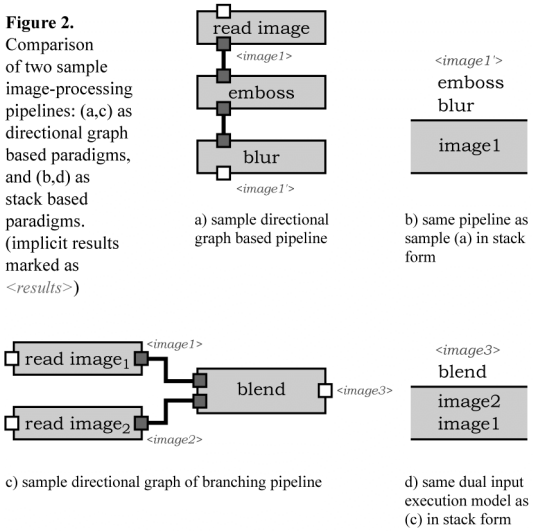


Figure 1. Baseline, action stacking, action history

We must first differentiate *data* from *parameters*, as they will be treated differently. We define data to be major streams or blocks of content that are to be processed, in order to create new sets of data. Parameters on the other hand tend to be much smaller information sets which characterize how operations are performed. From the execution model point of view, data streams from beginning to end across an action module until finished. A parameter meanwhile is always persistent across a particular action module. If a parameter changes value it can either trigger execution, or it can interrupt current execution, to restart a new one. This is task and execution model dependent.

The suggested paradigm operates as a metaphor of 'stacking' actions on top of objects placed on a table top.

Figure 2. Comparison of two sample image-processing pipelines: (a,c) as directional graph based paradigms, and (b,d) as stack based paradigms. (implicit results marked as *<results>*)



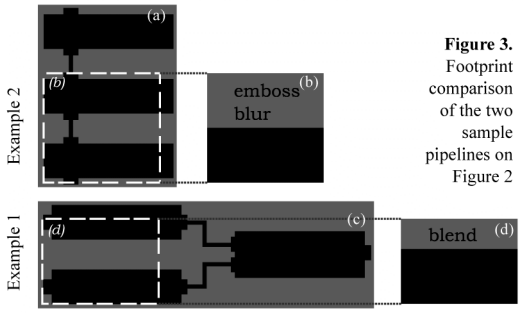


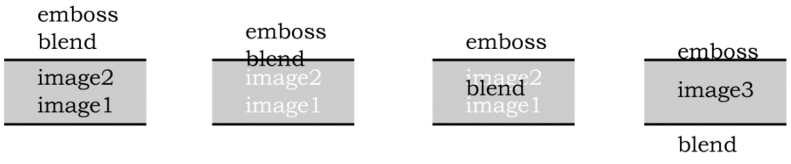
Figure 3. Footprint comparison of the two sample pipelines on Figure 2

We first define the stack *baseline*. The baseline is in fact a "dock" for data object stacks. Objects placed on the baseline, are subject to actions or operations. As new data is generated it takes its place on the baseline.

Action modules, or filters, are then placed directly on top of objects. Action modules will act on the objects on the baseline. Figure 1 depicts the basic visual layout of objects and action modules, Figure 2 illustrates examples of actual filters both under a directional graph environment as well as under a stacks environment. Notice already in these small examples the difference in space footprint used, as depicted more clearly in Figure 3.

We define a *script* to be a set of actions stacked to accomplish some task, or more simply, as a arrangement of actions and objects. Upon start of execution, the actions module will travel across the baseline–assuming the right number and type of inputs is available–affecting the data. Once the action module is finished executing, the resulting

Figure 4. Sample execution sequence of one action across time on two input objects



a) original placement of data objects on baseline

b) execution started. "Blend" action acts on objects

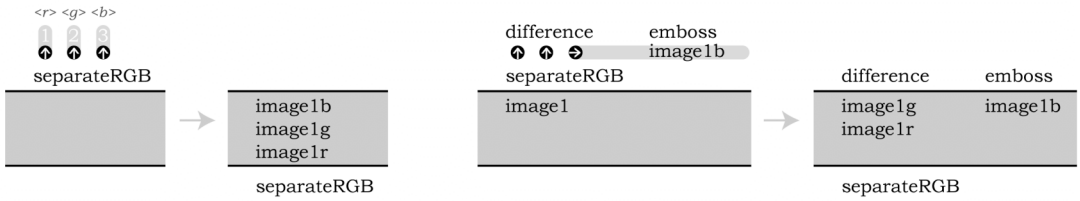
c) action passing across base line animations helps track work progress

d) result is put on baseline, filter is put at bottom as a filter history

`Baseline.-`	Dock for data objects. Top of data stack.
`Action Module.-`	Atomic action or filter.
`Data.-`	Stream or block of information to be processed.
`Parameter.-`	Attribute which characterizes the execution of an action module.
`Script.-`	Net of data objects and action modules.

Table 1. Definitions

Figure 5. multiple outputs, and output redirection



a) action module resulting in three output data objects placed on baseling (implicit results marked as <*results*>, push order marked with n)

b) action module, with two output data objects placed on baseline, one horizontally pushed onto baseline

data are placed on the baseline, while the action module itself is placed underneath the baseline, as a historical reference, and to preserve the consistency of the visual scrip, as exemplified in figure 4 using simple image filters.

Figure 6. Sample horizontal push copy, with remote invocation down the line (implicit stacked results marked as <*results*>)

The methodology so far depicted can be used to script a linear pipeline in an easy, intuitive, efficient, and compact method. However, the paradigm needs to be further refined to allow for multiple inputs, multiple outputs, and complex execution threads, such as loops, jumps, encapsulation, and conditionals.

4.1 Multiple Inputs/Outputs

Multiple inputs and outpus are basic premises under all stack-based languages. Operations requiring multiple input objects simply pop as many elements from the top of the stack as needed (see figure 2d.) And whenever multiple outputs are generated, these are simply pushed to the top of the stack (see figure 5a.) We follow on these same notions on our visual system.

As an illustration consider the following two simple algebraic expressions, with the corresponding results:

`a + b → c` and `f (a) → (b,c)`

their translation into a postfix (stack based) expression would correspond to:

`a b +` and `a f()`

with the corresponding results on the stacks being:

`c` and `b c`

To avoid ambiguity when dealing with multiple inputs, we first mark outputs in order from left to right. This will be used more upon implementation as a visual tool with individual output glyphs. But in order to allow for access to a particular object resulting from an action, we define a "horizontal push" as seen on Figure 5b. The basic premise is that when multiple outputs are generated, results are pushed on the same stack. When an output should not be pushed on the same stack, but should form a new stack, we allow the user to symbolically mark a horizontal push of a particular output object through the use of the *horizontal push fork*, which is then uniquely labeled and forms the new stack.

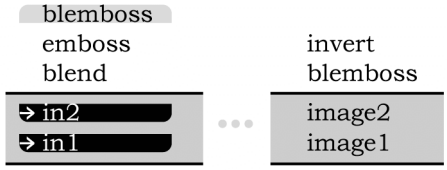


Figure 7. Encapsulation example and sample call

4.2 Remote Data Invocation, Encapsulation

We further expand the concept of a horizontal push to be used in any situation when the current state of a data-object needs to be duplicated or a fork is needed in order to split into a separate action stack. We use this notation to encompass remote invocation of results or data from different stacks as seen on Figure 6.

We then extend the system to allow for undeclared data references, and group naming of action stacks. With this we make it possible to procedurally call action stacks with specific data objects being passed in. We do this by a simple group operation, which creates a unique name for a set, while the input data is marked as undefined (place-markers). Notice that the already defined way input and output are used and placed on the baseline, (stack-top) is enough to define requirements for inputs and outputs. For an example see Figure 7. The actual procedure for labeling, markup, and grouping is discussed as part of the implemented environment, Section 5.

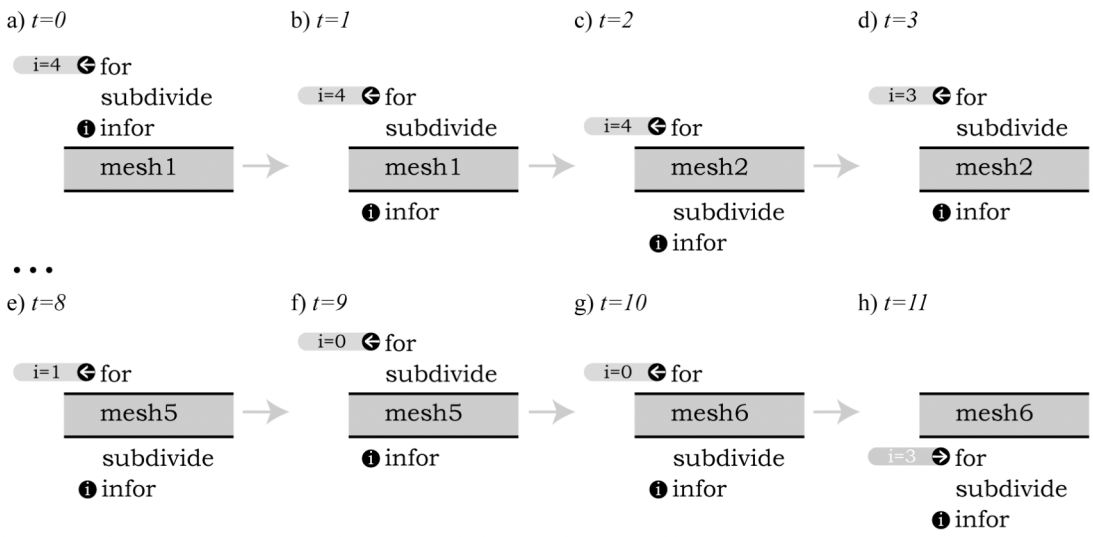


Figure 8. Sample sequence of loop execution across time (t=0 to t=11), for counter value i, from *i=4* to *i=0*, resulting in five levels of *subdivision* on initial object input *mesh1*.

4.3.1 Loops, Jumps and Conditionals

Loops are the most commonly used control switch in visual systems. Rarely are conditionals used, while direct execution jumps are usually not available. Yet switch statements are the key to enabling simplification of complex programming threads. Though we will mostly focus on simple for loops, we will define conditionals and execution jumps as building blocks for the more intricate for loop.

Conditionals allow a single parameter to be evaluated to choose amongst two paths of execution. When a condition is met, we continue vertically with the execution path, while if a condition fails, we move execution to a right defined action path. Since this represents a move of action stacks, the baseline, and its internal object stack, remain the same, and will be available to either chosen execution path.

Jumps meanwhile allow for a direct jump from one part of the execution thread to another. If the jump is within the same continuous vertical execution stack, then the missed actions are simply skipped over. In the case where the jump target point is on a different stack, the targeted stack, and its above contents are copied over to the current stack. Jumps are marked in a comparable way to how data horizontal pushes are symbolized. We define *jump-out tags* and unique *jump-in tags*. An instantiation of a jump-in tag among a stack of actions is always ignored on normal execution. Meanwhile the presence of a jump out tag forces a "jump" in the execution.

As such, loops are a simple combination of both of conditionals and jumps. A *for-loop* is a specific case of a loop, that carries the value of a particular parameter, repeats a particular set of actions decreasing the parameter's value every occurrence, until the counter is exhausted, and execution is then passed outside the loop. We exemplify how a *for-loop* is instantiated under the stack-based system, as can be seen in Figure 8.

4.3.2 Semaphores

An alternative to the proposed switch operations explained in the previous section can be to define basic semaphores to produce similar execution threads. A *wait* operation halts the action of a current stack, as it waits for some specific tagged data object to be generated. If the semaphore wait is coupled with remote calls of objects that would be on the stack, this would be equivalent to a jump operation.

Semaphores are simple to introduce and are a logical approach to some basic execution strategies. But overall, they are un-intuitive and prone to creating deadlocks that will freeze execution. Thus, loops, conditionals, and direct jumps will be easier to utilize for novice users.

Figure A1 shows an example where semaphores were used as a need resulting from the use of operations that take indiscriminate number of inputs (i.e. pop the whole baseline stack.) If such operations were not permitted (any action module had a determined number of inputs), then the three large stacks could have been grouped into one larger stack with the for loop around it.

5. The Environment

A stack based visual framework for visualization, as described in the previous section, was implemented using Processing [Fry]. The basic symbology and presentation of the interface was refined to operate in a pixel-based medium. We utilized direct means of interaction to maximize user efficiency and understanding of the system.

- Users can interactively populate the base line with simple drag-and-drop actions from a data file repository (data file manager).
- Actions are then stacked on top of the baseline as chosen from a action module repository (interactive list).
- Right push copies are done with a simple shift-click.
- Groups are defined with a selection lasso, where inputs are then generalized as parameters

Currently the library of action modules includes wrappers to ImageMagick, filters for image manipulation. The execution does not place direct calls, but rather python scripts dumped from the interface, as described in the visual framework models of [3].

Animation is both present upon execution as well as a set of discovery tools to quickly highlight remote object references or jumps.

6. Results and Conclusions

The images in appendix A describe a complex execution script for an actual map used to visualize data at the ASC/ASAP Center at Caltech. The figures include a side-by-side comparison of a directional graph implementation using NAG's Iris Explorer versus our stacks based representation.

We have presented a new methodology for visualization systems based on stack-based languages and object-based programming. We have presented the rules to govern such a paradigm, as well as an actual implementation. We believe its compact implementation, and intuitive nature will prove an effective solution for visual frameworks.

The use of animation, direct drag-and-drop features, as well as additional tools for thread discovery and manipulation are extremely helpful to aid in the usefulness and ease of use environment. Furthermore, the animation of the execution itself is one of the strongest indicators to aid in the understanding of how a task is being accomplished. Similarly, the ever-present baseline containing data objects, and especially intermediate states of data before the end of execution is reached, is particularly helpful in following how work progresses and as a debugging tool with access and a window to results from internal steps and actions.

We acknowledge that this paradigm works best on sets of filters that mostly have a very small number of input and output data objects, which is actually the case for most scientific visualization we have encountered. The same is true for directional graphs.

We have encountered, that when execution branching occurs, a stack system shows more orderly arrangements that a directional graph based system. However, if branching is extensive (including conditionals and jumps) then the better alternative will be a pure text based scripting language.

8. Future Work

Our first immediate goal is to completely import the visual stacks paradigm and current implementation into the more portable and flexible VPE framework. From there, external engine coupling should be much easier and better defined, so that we can easily write wrappers around VTK filters. It will then serve as a fast visual scripting solution for visualization producing scripts that may actually be compatible with an alternate visualization system and parallel renderer such as Paraview.

We further have a commitment to enable this system as a visual framework for both the ASC and DANSE projects at Caltech. Through such, we will be able to gain direct experience as to how all aspects of our interface perform. Furthermore it will give us opportunity to explore additional uses for the interface. For example, the DANSE researchers are currently looking for better ways to sort and query through large databases of data, to then select sets of data to feed to the visualization framework. Meanwhile, the ASC researchers are looking for ways to have the visual environment encapsulate information on the state of a simulation is being executed across a large parallel system, giving visual feedback (i.e. visualization of mid-steps) that can be helpful for simulation debugging and steering.

We also believe it will be useful, once Python hookups are available for our VPE interface, to use the available wrappers around VTK filters to make them available through our stack based interface. This will serve as a fast visual scripting solution for complex execution models, possible producing scripts compatible with Paraview which can be used as a last step interface and parallel renderer.

We have done some initial work into extending the stack paradigm into a physical or tangible interface. Action modules as well as objects will take the form of actual physical objects or *bricks*. Bricks start as empty cases, which through a gesture-based interface (based on the environment) can take form of an object from a file system or an action from a library. The stacking action will be the same as currently described, though special treatment has to be made for horizontal data movement. We are currently working on clarifying interaction issues, as well as objectively measuring the effectiveness of different techniques and the overall sense of control and presence.

8. Acknowledgements

This work was funded by the ASC/ASAP Center of Excellence at the California Institute of Technology and Caltech's Summer Undergraduate Research Fellowship. We would like to acknowledge Michael Aivazis for his collaboration and support, as well as Mathieu Desbrun and John McCorquodale for help in the writing and editing of this work.

References

1. Ben Fry and Casey Reas "Processing Code." *Aesthetic Computing*. Paul Fishwick Editor. MIT Press, Boston, 2004.
2. Daniel D. Hils, "Datavis: a visual programming language for scientific visualization", *Proceedings of the 19th annual conference on Computer Science*, pp 439—448. ACM Press. San Antonio, 1991
3. T. J. Jankun-Kelly and Kwan-Liu Ma, "A spreadsheet interface for visualization exploration", *Proceedings of the conference on Visualization '00*. pp.69-86. IEEE Computer Society Press. Salt Lake City, 2000.
4. Santiago Lombeyda, Michael Aivazis and Mahesh Rajan. "Making Remote Tools Available for Visualization of Large Data Sets", *Proceedings of the Visualization Development Environments Workshop*. Princeton Physics Laboratory. Princenton, 2000.
5. Fabian Nunez, Edwin Blake, "ViSSh: A Data Visualisation Spreadsheet", *Data Visualization 2000*. pp.209-218. Springer-Verlag/Wien. Amsterdam, 2000.4.
6. Peter Wegner, "Dimensions of object-based language design", *Conference proceedings on Object-oriented programming systems, languages and applications*. pp 168—182. ACM Press. Orlando, 1987.

Table A. Rules for stacks paradigm

i. The stacks environment is by definition an active and dynamic space, governed by basic rules. The space is defined by:

a. The space is subdivided by the *base line*, where only objects (or references to objects) can exist. The baseline is a stack in of itself.

b. Above the baseline *action modules* are stacked.

c. Below are actions that have already been executed.

ii. *Actions* fall and execute, as by gravity, traveling across the baseline given the necessary *inputs* are present on the baseline. Actions *pop* used objects of the baseline, and *push* results back onto it.

iii. *Multiple outputs* coming out of an action module are marked (upon request) in left-to-right order, same as order as they would be pushed onto the baseline.

iv. An output can be tagged and redirected onto the same stack, or to a new stack as a *reference*.

v. Objects or references to objects can be placed on a stack. Undefined references halt the actions until they are define.

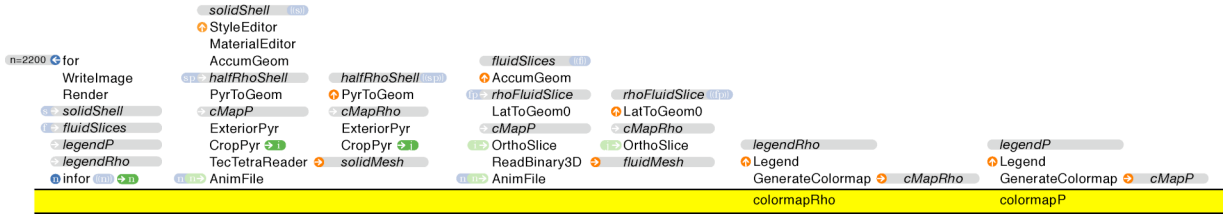
vi. A *jump* is used as a one directional jump in execution. It forces a virtual a copy of actions on the stacked above the *targeted jump point* to be placed instead of the currently active actions stack.

vii. A *conditional* is a switch between two possible execution stacks. The chosen stack takes the place and executions upon the current baseline.

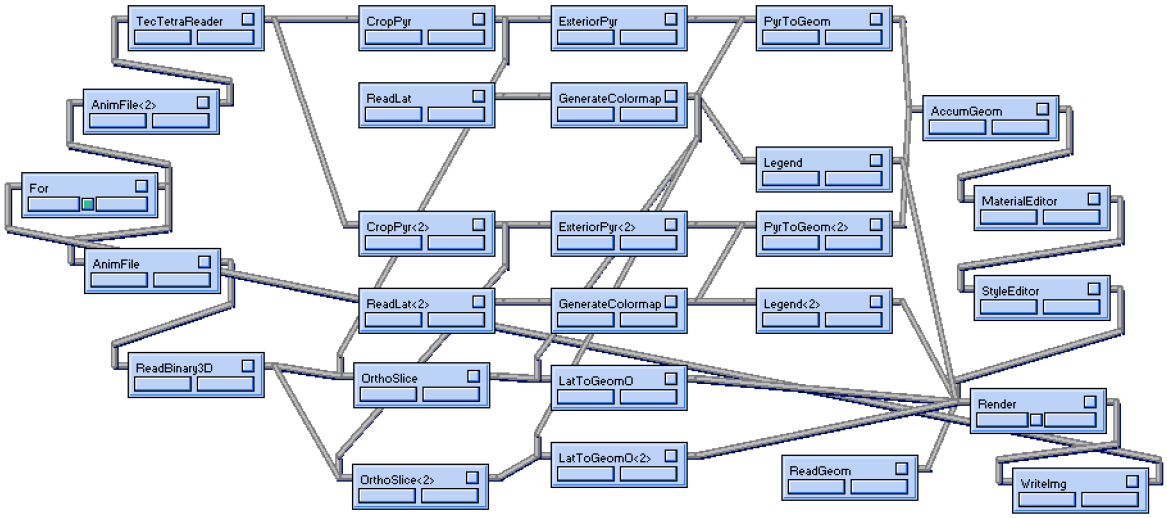
viii. *Semaphores* are tags that force a halt in execution. Execution will only restart once a complimentary *broadcast* is received. Reciprocally, a *broadcast* is a tag that triggers action across any complimentary semaphore(s).

ix. A *for loop* is defined be repeated set of actions, marked between an *infor* and the *for*, for a counter ranging from 0 to a set value.

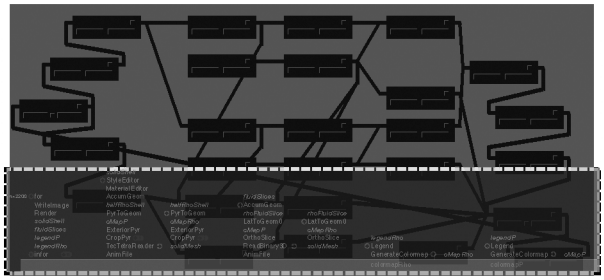
Figure A1-3.



A1) Screenshot of actual script from Virtual Test Shock simulation at Caltech in stack based visual framework. Script works around the use of two action modules, Render and AccumGeom, which both take as many inputs as available on the stack. That forces sub-processes to be forked into different execution stacks. This is done through the use of semaphores, as explained in Section 4.3.2. This script also shows examples of a for loop for 2200 steps, several push copies, vertical referencing of output objects, and remote object calls. Notice that on execution of the script, there are three stacks that can run immediately: the for loop and both the colormap-legend creation stacks.



A2) Screenshot of actual same script as above, using NAG's Iris Explorer under Windows environment (under Unix based environments maps a bit less compact).



A3) Difference in footprints between both scripts. Both maps are scaled by 50% from their original screenshots, and overlaid on top of each other. Notice that the stacks environment explicitly defines objects that correspond to information passed between pipes, but unseen, for the NAG's Iris Explorer version. Meanwhile, in the NAG's Iris Explorer map it is possible to rearrange the modules to have less internal space, however this sacrifices readability of the map. The map was manually arranged by a real user attempting to both minimize the space used, while preserving clarity of the task being performed across the directional graph.