2D Editor

An experimental editor for programmers using diagrammatic syntax (2D) can be seen here:

https://www.youtube.com/watch?v=8vZ8Pi32oMo

The experimental editor was unfinished, but demonstrated many important points.

The video has no sound. It demonstrates the beginnings of an IDE for a page layout editor – many of the concepts are related to generalised 2D editing.

The editor consisted of a “canvas” for drawings and a bank of buttons related to the application.

The 2D editor supported two kinds of diagrams – component-based flows and hierarchical state machine.

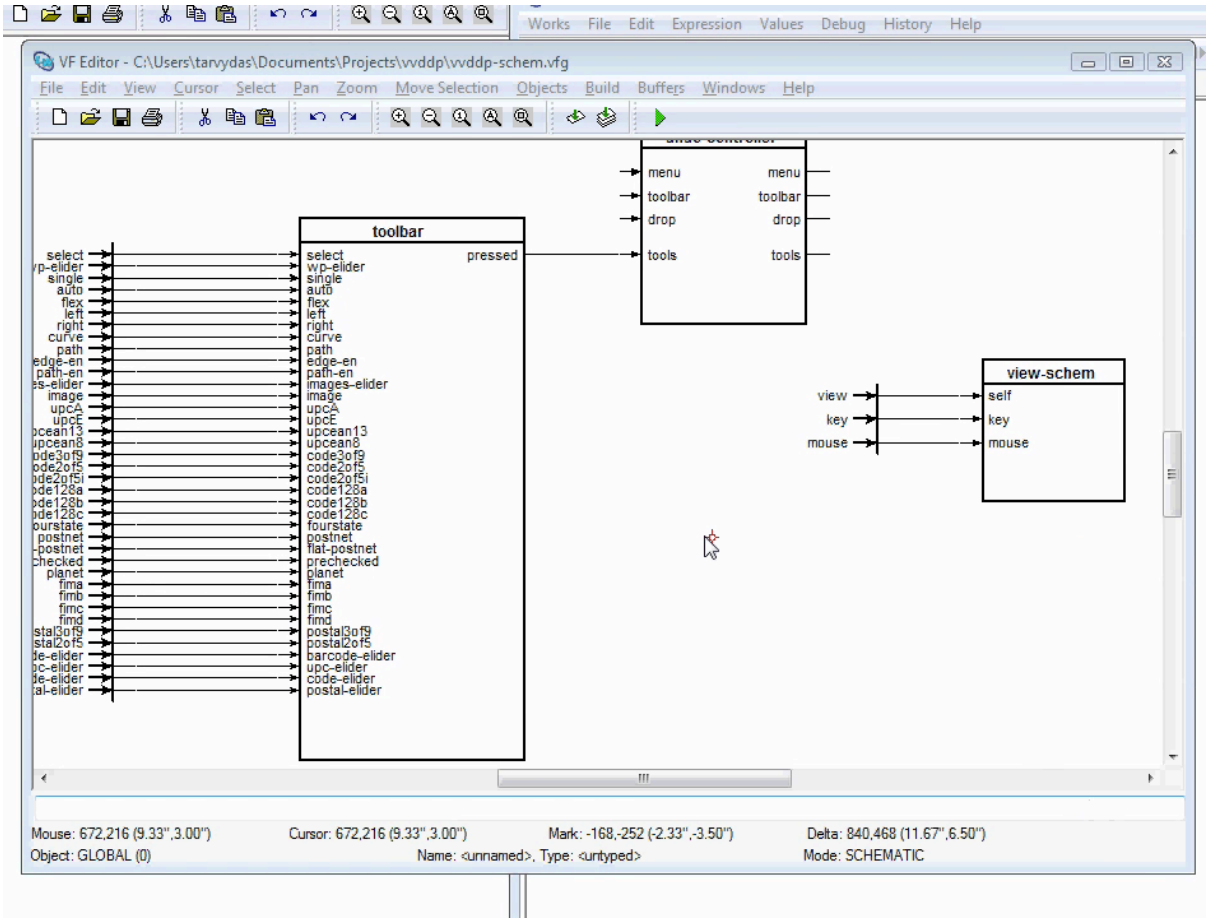
Towards the end of the video a live run of the application is shown.

The text below discusses some of the most important frames from this video.

TO BE DISCUSSED:

- keybinding
- objects (FB?)
- canvas
- Push buttons
- Undo
- indication
- roll-hits
- Parts, HSMs
- cursor/mark
- multiple buffers and views
- Part templates built from drawing primitives, then “normalised” (to (0,0)) and saved to a template directory
- grid
- the editor must not “know” what it is editing - it must simply allow drawing of graphical objects and must allow the drawing to be edited in a human-efficient manner (e.g. it should work as well as a code editor, instead of a VISIO(say) diagram editor)
- the most basic graphical atoms are: straight line segments, curved line segments, ellipses and text. (Note that rects can be drawn using only straight line segments, although, it might be “more efficient” to allow rects to be atoms).

Part Based Events



This is a “top level” diagram of the experimental architecture for a 2D diagram-based editor. The experiment was not put into production - this version is the latest revision. At this point in the design it was expected that three Parts would be needed. A “toolbar” controller, an “undo-controller” and a “schematic viewer” (view-schem).

A rectangle represents a Part. There are 3 Parts on this diagram - “toolbar”, “undo-controller” (chopped off at the top) and “view-schem”.

Each rectangle has a “title bar” which is at the top of the rectangle delineated by a horizontal line with a single piece of text above it (and completely inside the rectangle).

The “title bar” contains the name of the *kind* of the Part. A *kind* is like a *class* in other object-oriented languages. In other languages, it might be called a *type*. *Kinds* are slightly different from *classes* and *types*, so we use a new word *kind*. Each part on a diagram is treated as a separate instance of it’s kind.

This is an “inside view” of a schematic (Composite) Part. The schematic as 3 Parts

with very little wiring between them. There is a lot of wiring from the “outside” to pins on the toolbar Part. There are 3 inputs from the “outside” wired to the inputs of the view-schem Part.

Inputs from the “outside” are shown as bold, short arrows, terminating on a vertical line. There are no outputs to the “outside”. If there were any outputs, they would be shown as arrows going into bold vertical lines, followed by bold horizontal lines. See the “push-single” output Pin in Frame 3. In our current Draw.io drawings, we use ellipses to represent inputs from the outside and outputs to the outside.

Input pins are drawn as arrows touching the edge of Parts.

Output pins are drawn as arrows emanating from the edge of Parts.

In either case, the pin names are drawn as regular text near the pins at the edge of Parts and completely enclosed by the Part. In our current Draw.io drawings, we place pin names on the outside of Parts, near the junction of the arrows (arrowheads and arrow beginnings) and the Parts.

The *undo-controller* Part (top middle of diagram) has 3 N/Cs (No Connections) on inputs “menu”, “toolbar” and “drop”.

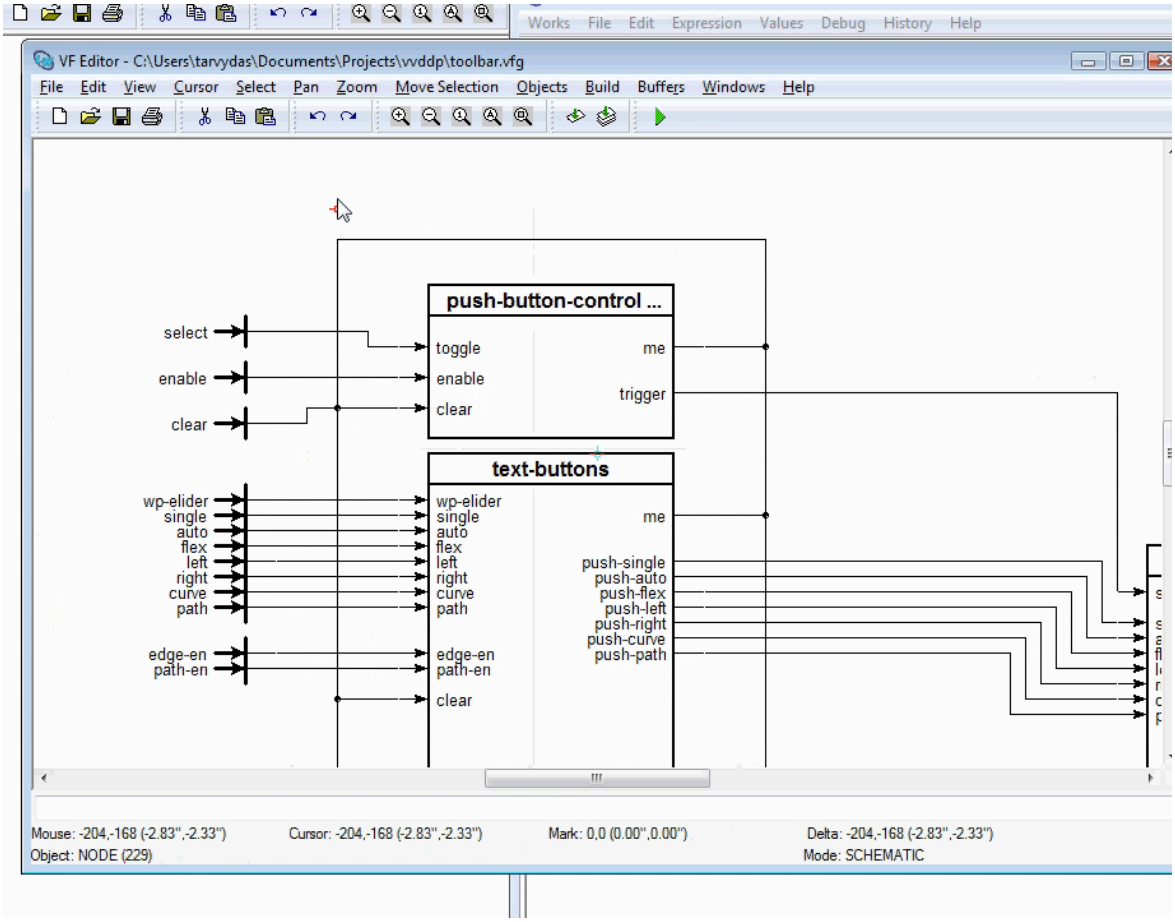
The *undo-controller* Part (top middle of diagram) has 4 N/Cs (No Connections) on outputs “menu”, “toolbar”, “drop” and “tools”.

There is one connection from Part *toolbar* and Part *undo-controller* - the arrow between *toolbar/pressed* and *undo-controller/tools*. Events flow from *toolbar/pressed* towards *undo-controller/tools*.

The mouse, in this version, has been bound to a function which draws a red cross-hair - the *cursor* - at the grid point nearest the mouse arrow. In the video, the mouse moves smoothly, but the *cursor* appears to move in a “jerky” fashion. This is due to the fact that the *cursor* is snapping to the nearest grid point instead of landing exactly on the point that the mouse is at.

TO BE DISCUSSED:
- cursor mark
- Lots of input ports go directly into toolbar. Looks like a lot of drawing effort, but a reasonable 2D editor makes this easy.
- what is toolbar, undo-controller, view-schem?
- Pin names
- port names
- status bar
- input API
- output API
- grid

Frame



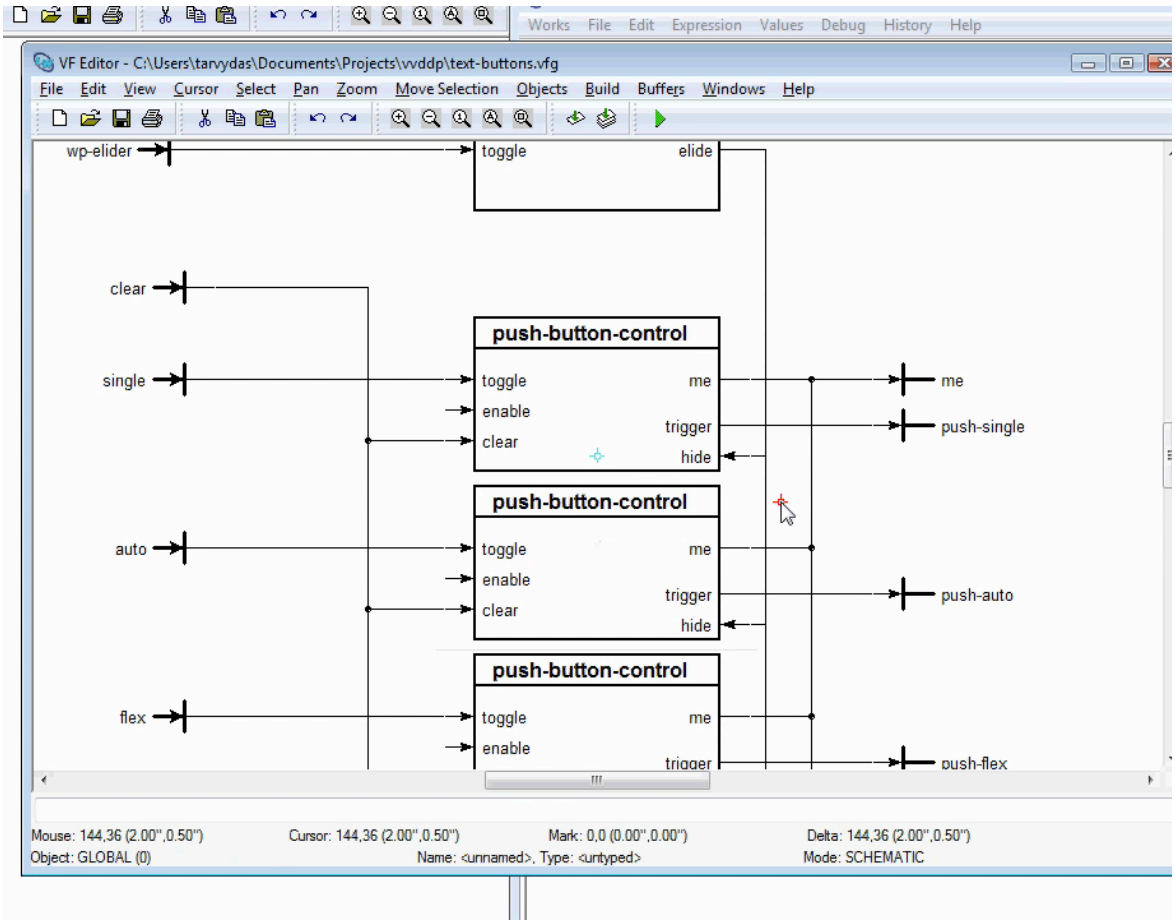
Editor pushed into toolbar Part.

Architecture: all buttons corralled into various buttons groupings. The only grouping visible in this frame is the "text-buttons" Part, which contains all of the buttons that deal with text in the application.

There are too many buttons to be displayed simultaneously. So, the design uses collapsable (ellidable) button groups within button trays. The Part, "push-button-control" controls the expansion and collapse of button trays. (See next frame for discussion).

TO BE DISCUSSED:

- dots
- "feedback" loop
- what is "me"?



Editor pushed even further, inside “text-buttons” Part.

Each button is a clone of the “push-button-control” kind.

Each button has six pins.

Outputs “trigger” and “me”.

Inputs “hide”, “toggle”, “enable” and “clear”.

The pictures and names of the buttons are handled outside of Bmfbp (in this case in LispWorks’ CAPI). All button-press events (from CAPI) are are converted to bmfbp events and funnelled into the “toolbar” Part.

Example: look at “auto” button. The CAPI button labelled “auto” is converted into a bmfbp event and arrives on the “auto” pin of this “text-

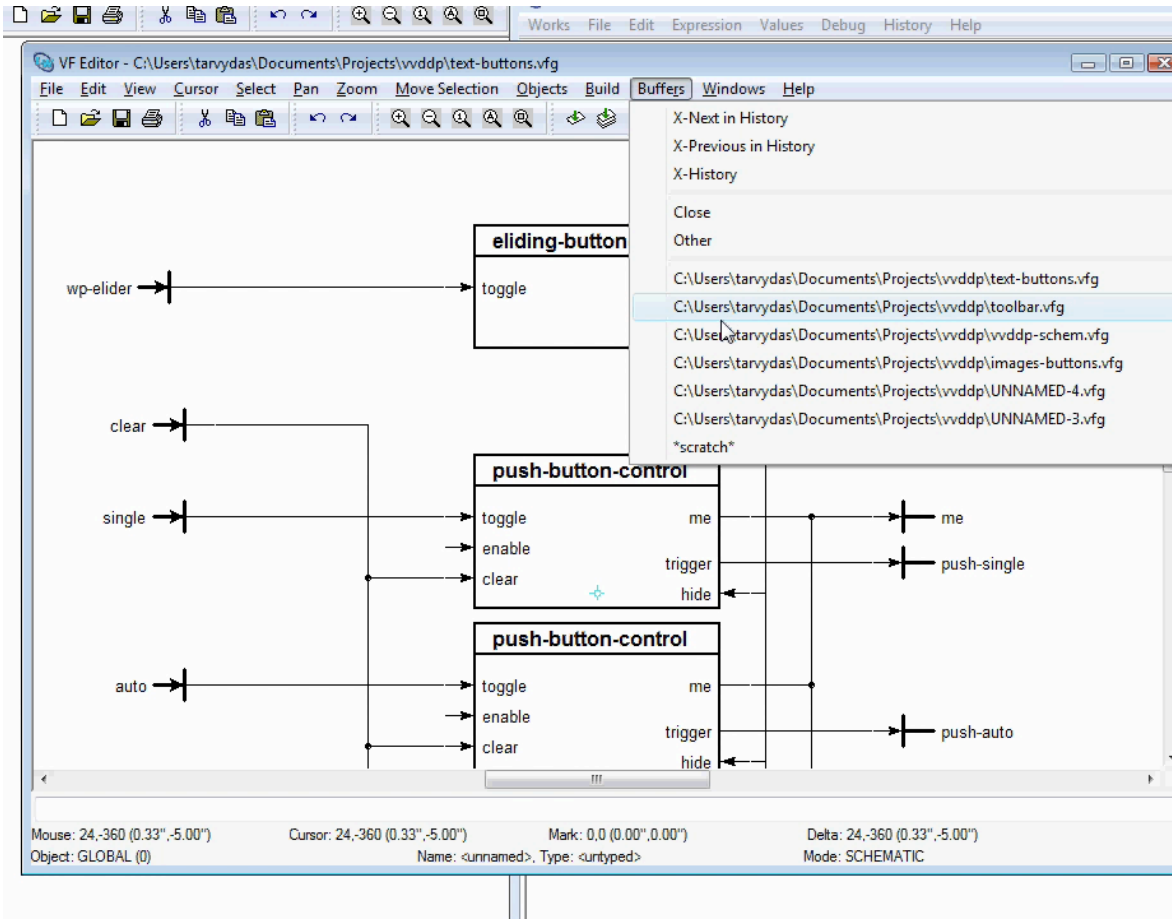
buttons" Part. The event arrives at a unique instance of a generic "push-button-control" Part, on the "toggle" pin. After some processing, an event is sent to the "trigger" pin. This trigger pin is connected to the "push-auto" output pin. In this way, each button is treated in the same way, but is labelled uniquely throughout the system.

The "hide" input pin tells the "push-button-control" part to hide itself (remove its image from the window). Note that the "hide" pins of all Parts are tied together and controlled by elide control Part (that is partially cut off at the top). The elide controller can tell all "push-button-control" parts inside of this Composite part to hide themselves "all at once".

The "clear" input tells a button to reset itself to the unselected state. In this architecture, all "clear" inputs are tied together, so that all buttons set themselves to unselected "all at the same time".

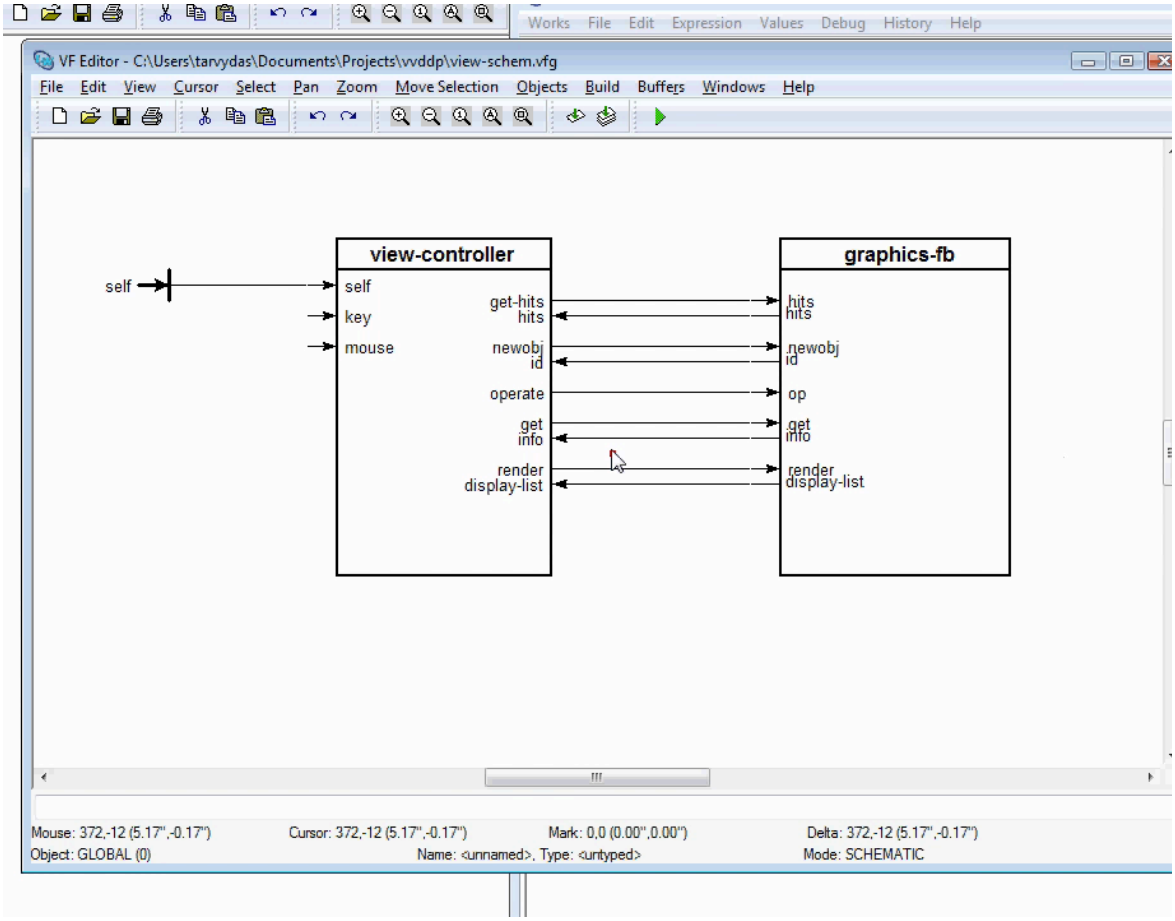
The "enable" button is unused in this architecture (it is mean to grey out a button and make is unselectable).

The "me" outputs are all tied together and are sent to the "me" output pin of the Composite. They are controlled by a controller outside of this Composite. I need to look at the spec for the "push-button-control" Part to remember what the "me" pin is for (this is a flaw in the expressiveness of this architecture).



This frame shows how this particular bound the buffer-switching command to a menu item. Buffer-switching might also have been bound to a keystroke, but this is not apparent here.

Each unique buffer holds one Part. This editor edits one Part per window. (It is not apparent if there can be multiple windows (views) for one Part).



Back to the top-level and "down" into the "view-schem" Part.

This snapshot shows the Architect's intent. There is a controller for the view and a FactBase that the controller controls. The "graphics-fb" Part is a server - it waits for a request, then answers.

A hit-test is a a graphic operation that determines if a graphical object (in the FB) is under the cursor. "Hits" returns a collection of such objects. More than one object might be underneath the cursor (see hit stack roll, later).

"Newobj" creates a new object in the FB and returns its (new) id (aka handle).

"Op" performs an operation on an object. The event contains the object id and a operator token. The specifications for operations can be found in the specs for "view-controller" and/or "graphics-fb".

"Get" returns information about a graphical object. The "get" event contains an object id, and the return value "info" contains a collection of information. Again, the meaning/contents of "info" are included in the spec for "graphics-fb"

The "render" event is simple a trigger. It tells the FB to render all of its objects and to return a collection ("display-list") of all objects that that were rendered. The architect chose not to show how rendering is done (at least at this level).

Hierarchical State Machines

Hierarchical state machines are much like Harels StateCharts, except without the concurrency aspects (which are covered explicitly by event-based flows and Parts).

A single state (drawn in this editor as a box with rounded corners) can contain *entry code* (at the top of a state), *exit code* (at the bottom of a state) and *state code* (in the middle of a state). Code, in this editor, appears as Text and is elided (to save layout space). Code can be examined and edited using some keybinding.

A small circle indicates the "start state" for a given machine.

Transitions between states are shown as curved line and contain a block of code. The first line in the code is the event that triggers a transition and the rest of the lines of code represent code (actions) to be executed during the transition.

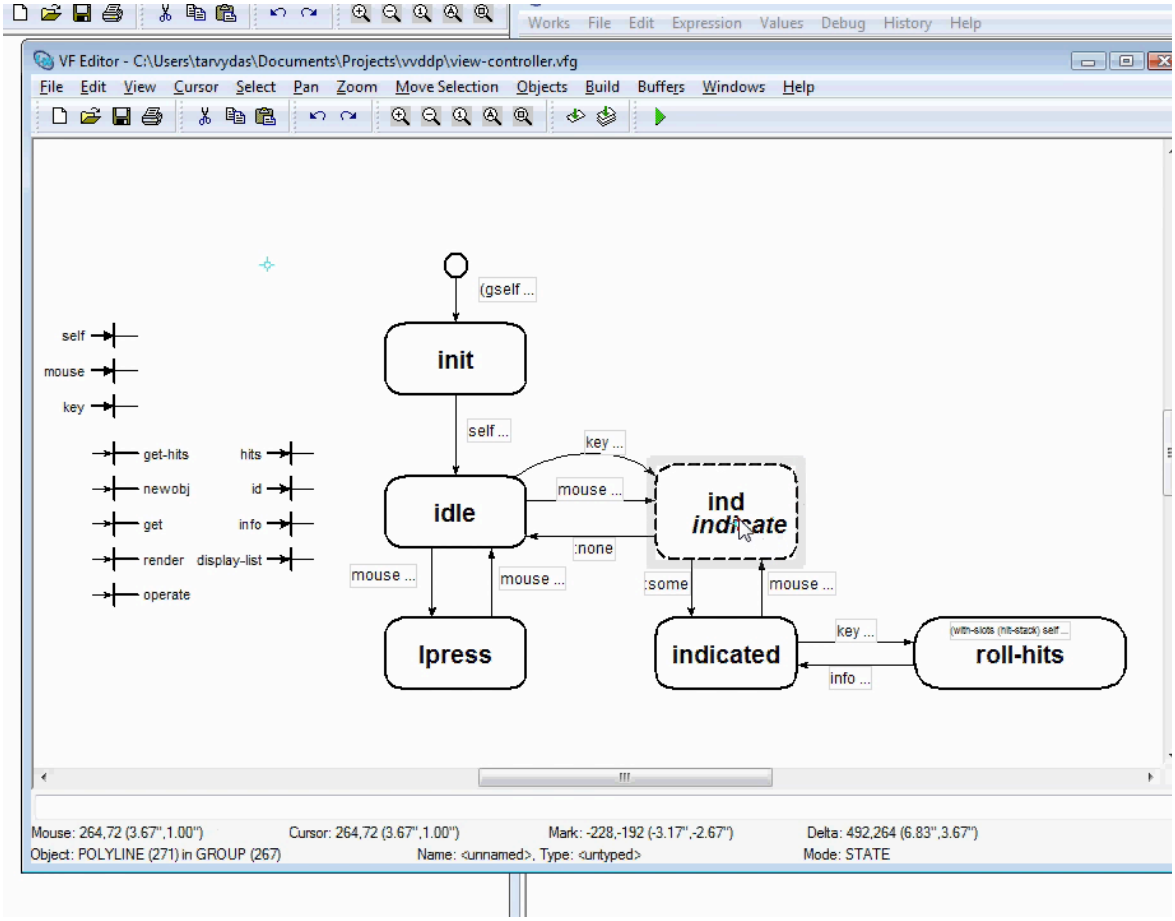
Sub-machines (children state machines) are drawn as rounded rectangles with dashed lines. Sub-machines have the same code (entry/middle/exit) as above.

Parent states always take precedence over children (the exact opposite of

OOP-based inheritance).

The editor must be able to draw hierarchical state machines, as well as event-based flows. The editor must not "know" what it is editing - the compiler, later in the workflow, determines if a diagram is legal (e.g. a flow diagram or a Hierarchical State Machine (HSM) diagram).

The editor can send tags/attributes to the compiler to suggest what type of diagram the Architect thought was being created.



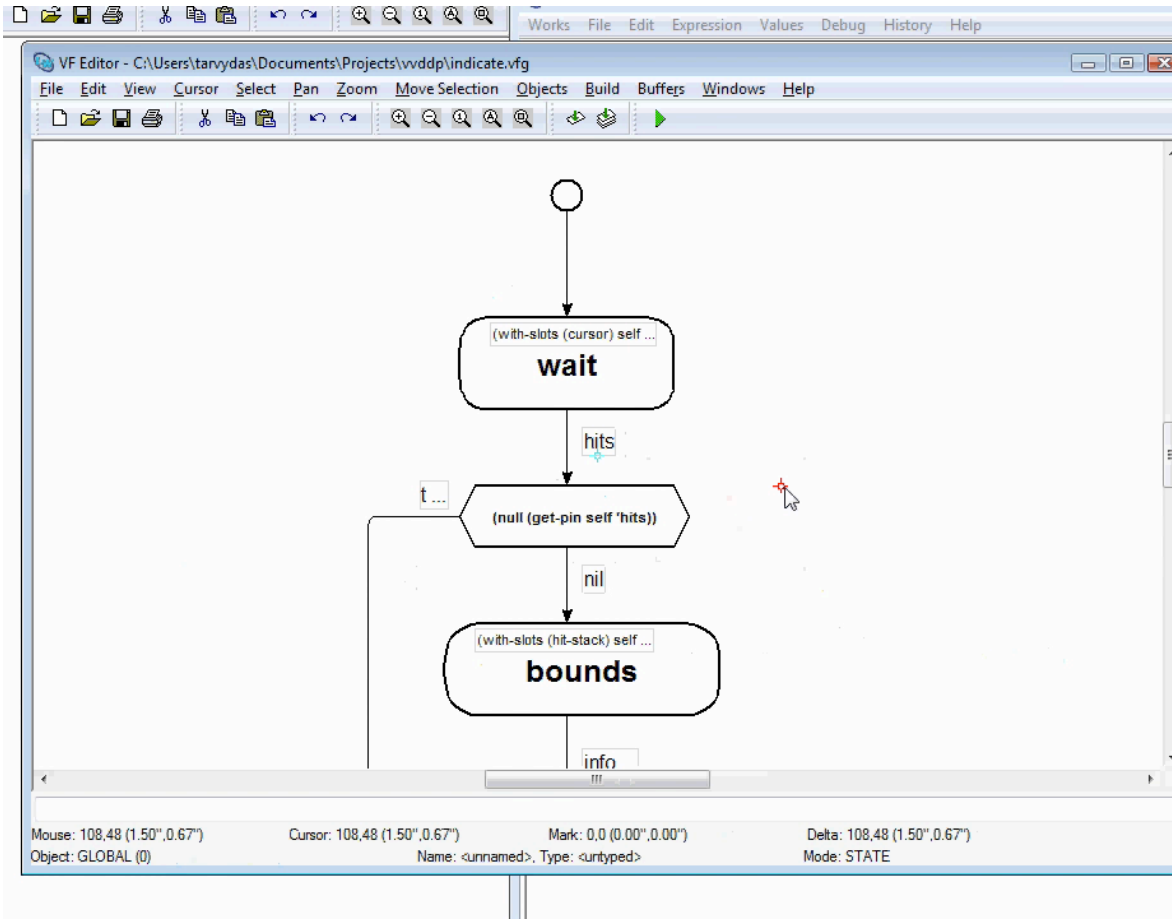
A fairly simple architecture for the "view-controller" Part.

This HSM starts in the "init" state. When it receives the "self" event, it moves to the "Idle" state. ("Self" is a handle to a View).

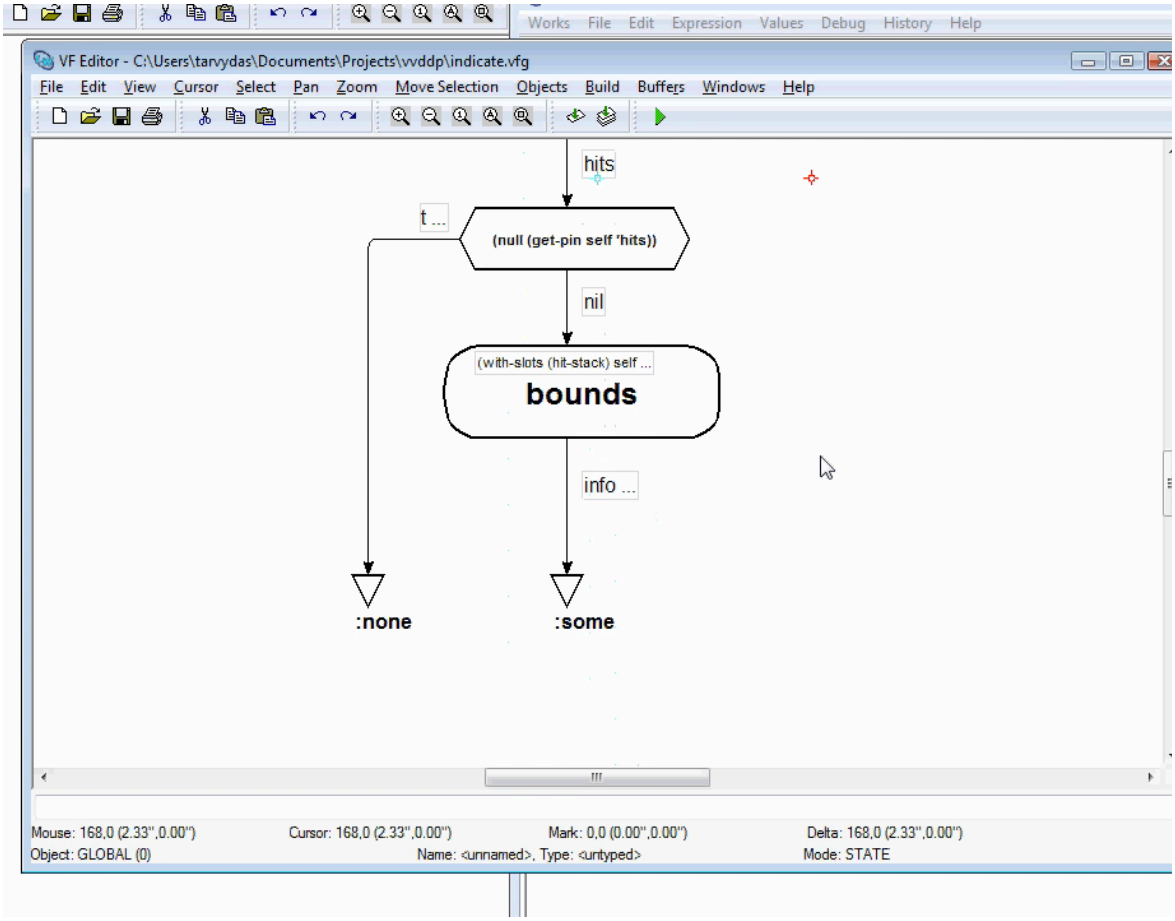
When a "Mouse-Down" event occurs, the HSM transitions from "Idle" to

"LPress".

When a "Mouse-Move" event occurs (in the Idle state), the HSM transitions to the "ind" state (the child is of kind "indicate"). The "indicate" HSM decides whether there are :some graphical objects underneath the cursor, or :none. If there are :none, then the HSM transitions back to Idle. If there are :some, then the HSM transitions to the "indicated" state. Indication is a concept taken from Jef Raskin's book "The Humane Interface" and is similar to changing the colour of a url-link in a browser as one mouses-over the link. In this design, an indication displays/lights-up the bounding box of the indicated object. Indication shows which object will be operated on (e.g. "selected") if the appropriate key is hit (e.g. left mouse down). From the "indicated" state, hitting a particular key (TAB, in this case) causes a roll of the hit-stack. Only one graphical object can be indicated at one time. If multiple objects are under the cursor, a stack (or queue) of objects is kept. The top object in the stack is indicated. If the roll key (e.g. TAB) is hit, the stack is rolled, the next object down becomes the top object and it becomes indicated.



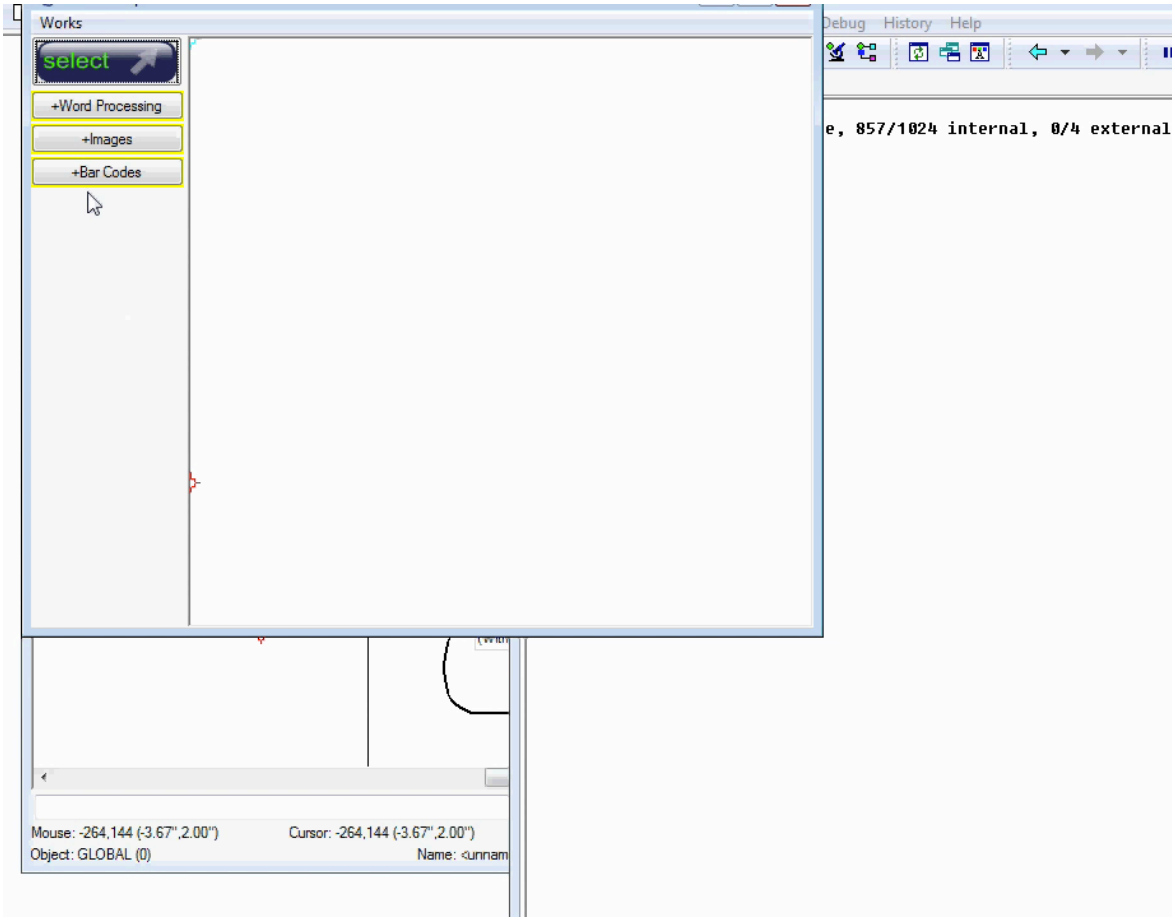
Some of the logic for indicating a graphical object.



The bottom part of the previous logic.

Downward-pointing triangles are HSM exits. These are like internal events. This child HSM tells its parent that either :none or :some objects are underneath the cursor. After sending such exit events, this child HSM resets back to the top. The parent receives these "internal" events and acts on them as if they were events from the outside.

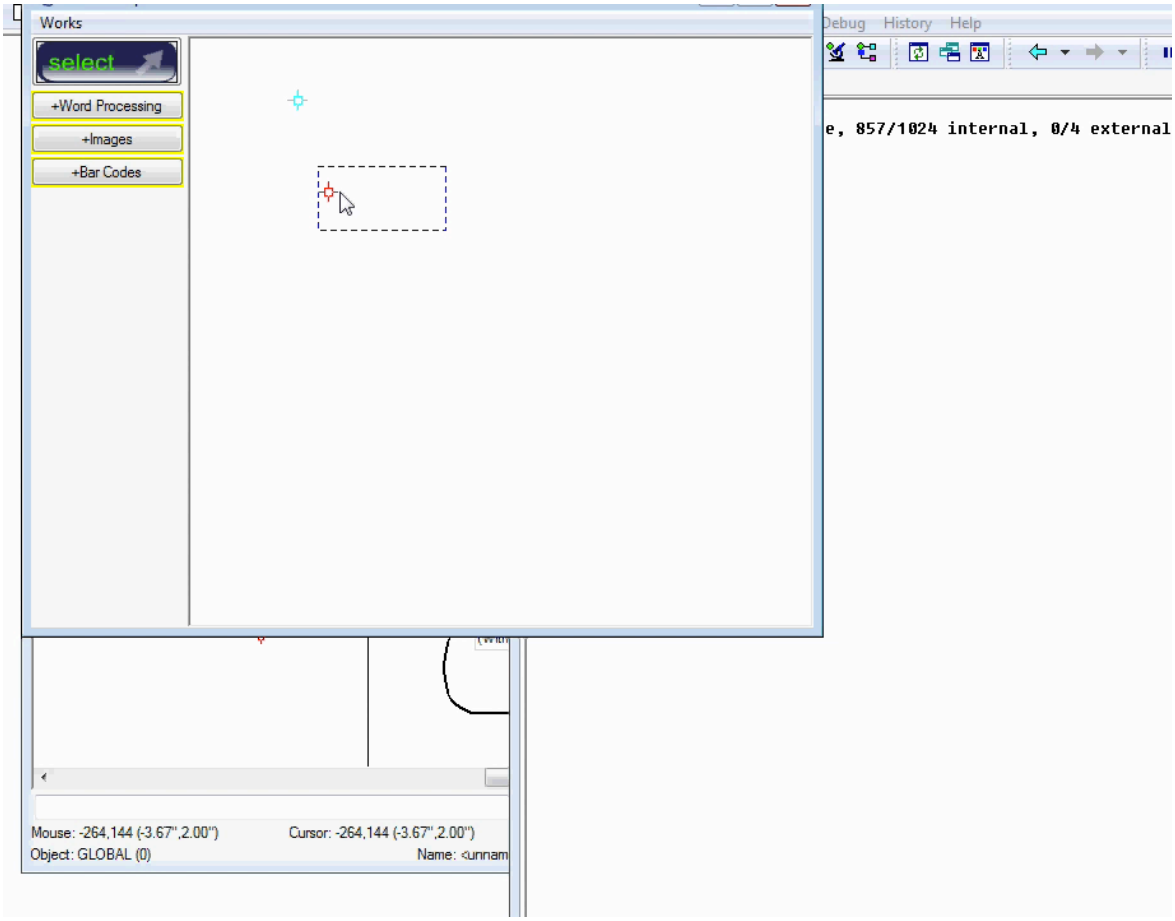
Live Run of the Unfinished Application



The smaller window at the top left is a live run of the application, built thus far.

Buttons on the left-hand side, a canvas on the right.

There are 4 active buttons at this point - Select, Word Processing (which opens a tray of word-processing buttons), Images (also a tray of buttons) and Bar Codes (also a tray of buttons).

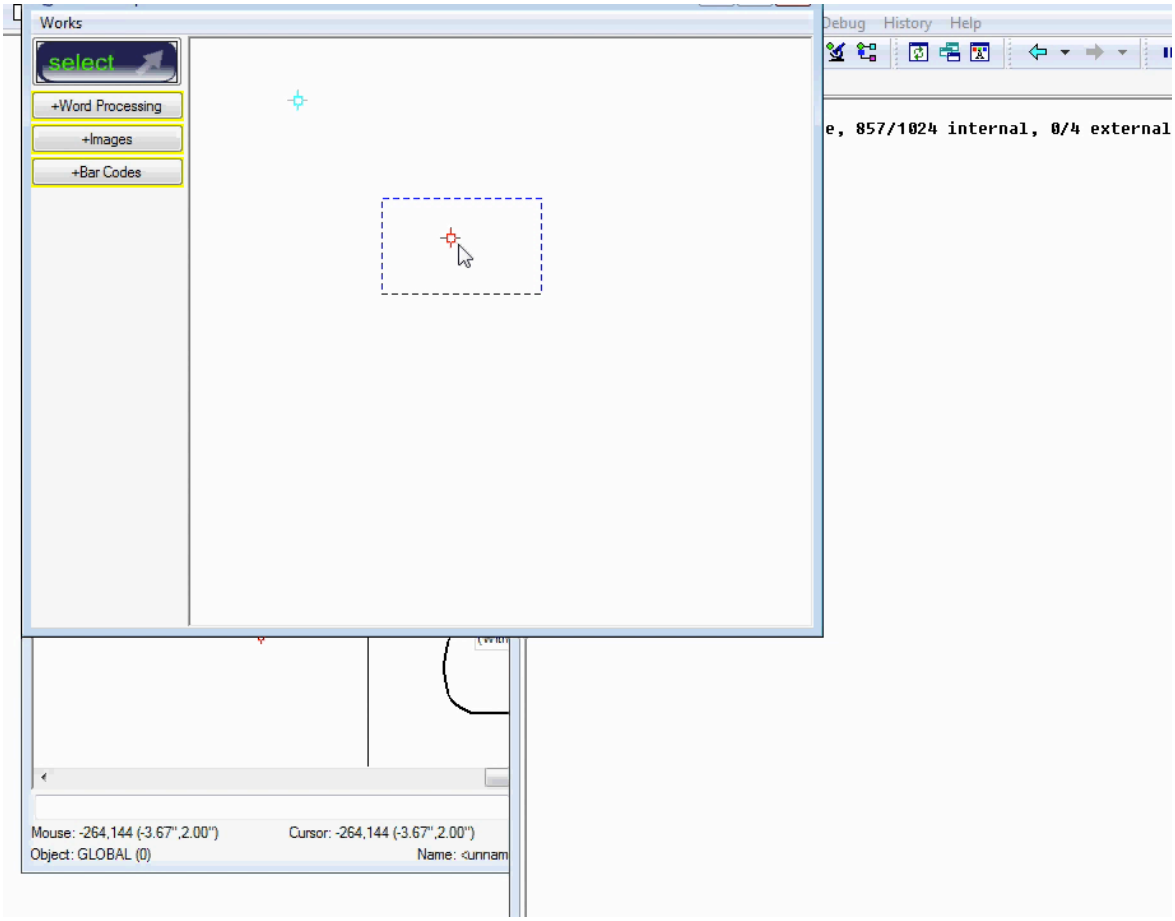


A mark (cyan crosshair) has been dropped (left mouse click at some empty point on the canvas).

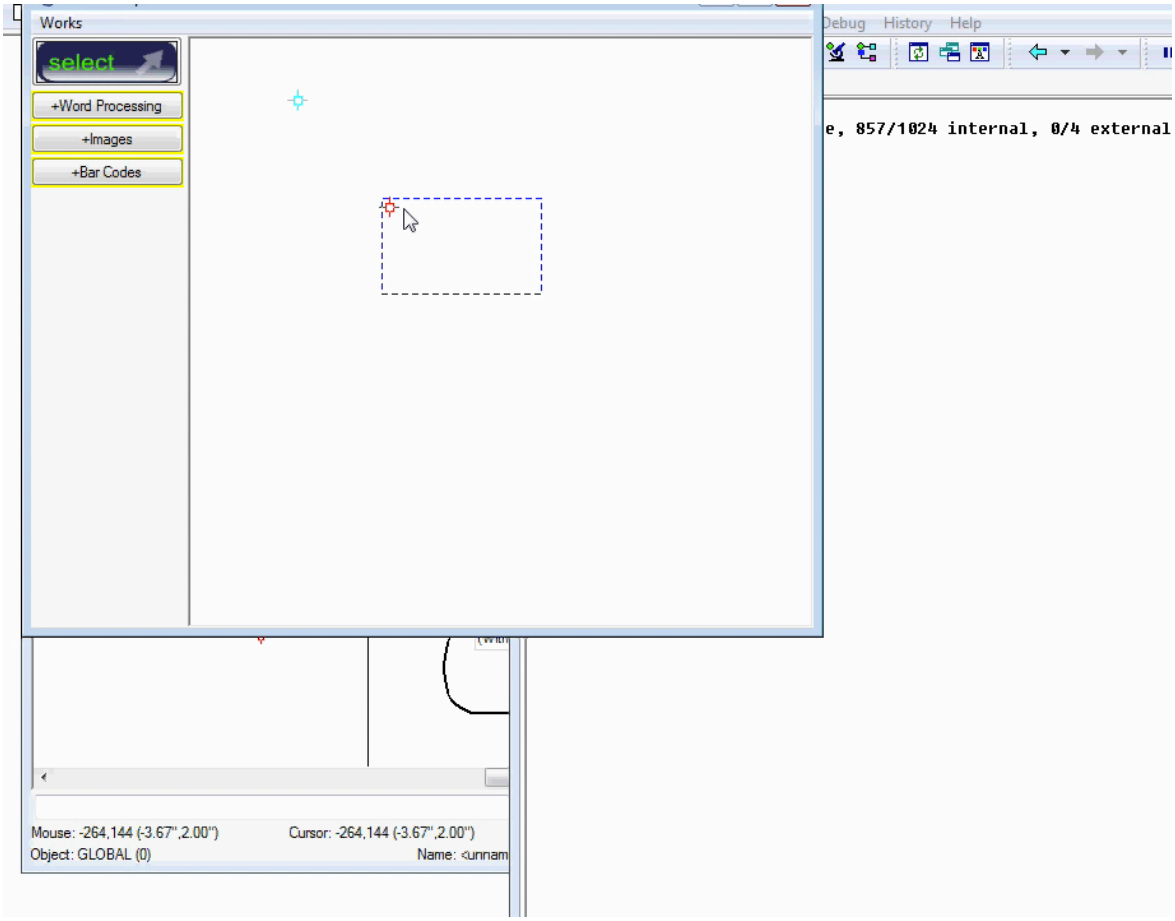
This application is unfinished (it was never finished). There are 2 objects in the factbase, but the objects are never rendered.

The cursor (red crosshair) is hovering over one of the 2 objects. The objects' bounding box is displayed as a dashed rectangle.

The other object is downwards and to the right of this indicated object. The bottom right corner of object 1 overlaps with the top-left corner of object 2 (object 1 is currently indicated).

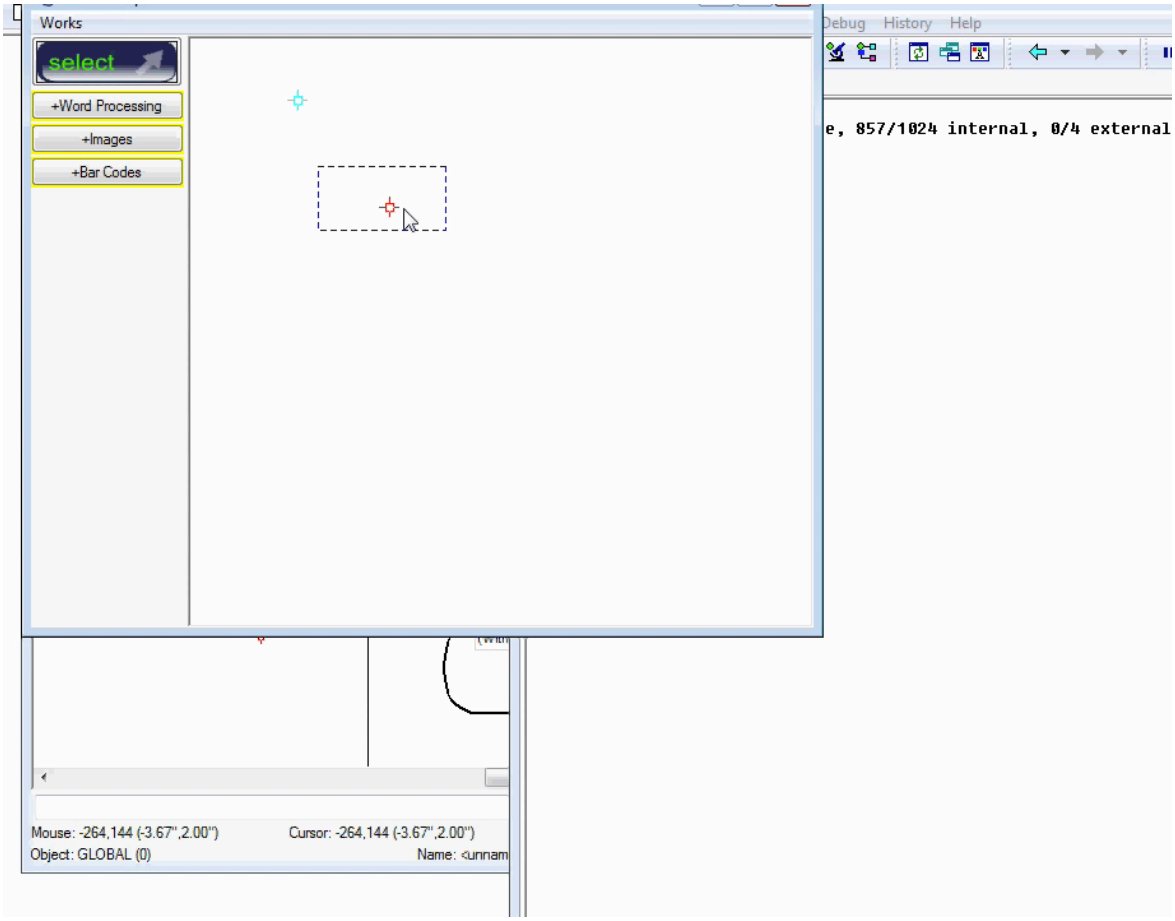


Mouse has moved and now it is hovering over object 2.



Mouse has moved to the top-left corner of object 2. This is the place where both objects are over-lapped.

Object 2 is at the "top" of the hitstack, so it (object 2) is indicated.



The TAB key has been hit. The mouse has not moved.

After the TAB key was hit, the hit-stack was rolled and, now, object 1 is at the top, hence, object 1 is now indicated. (If we hit the TAB key again, the hit-stack would be rolled, again, and object 2 would be at the top and indicated).