12. Visual Programming
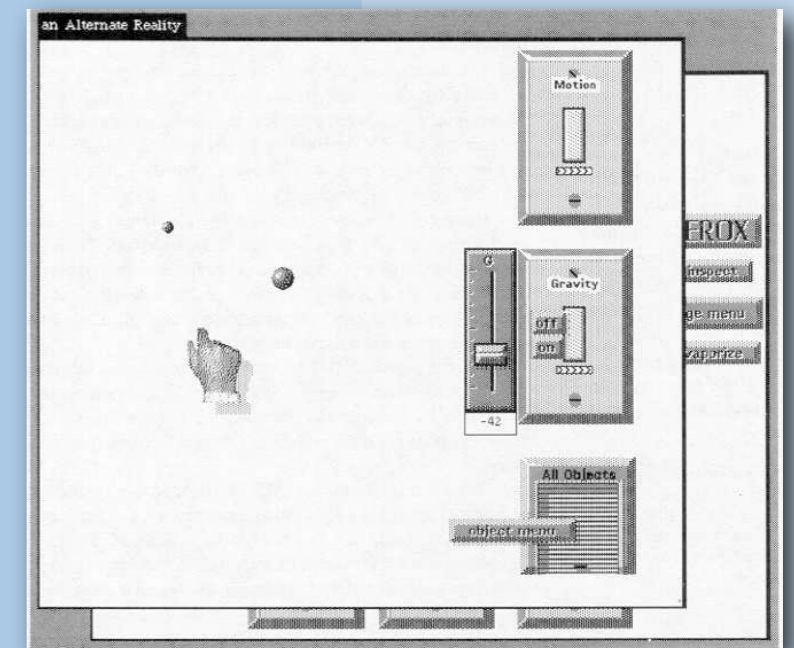
Oscar Nierstrasz

Roadmap

> Terminology
> A Quick Tour
> A Taxonomy of Taxonomies
> EToys (demo and evaluation)

Roadmap

> **Terminology**

> A Quick Tour

> A Taxonomy of Taxonomies

> EToys (demo and evaluation)

Sources

> Myers, "Visual programming, programming by example, and program visualization: a taxonomy," SIGCHI Bull., 1986.
  — http://dx.doi.org/10.1145/22339.22349

> Chang, "Visual languages: a tutorial and survey", IEEE Software, 1987.
  — http://dx.doi.org/10.1109/MS.1987.229792

> Burnett and Baker, "A Classification System for Visual Programming Languages," Journal of Visual Languages and Computing, 1994.
  — ftp://ftp.cs.orst.edu/pub/burnett/VPLclassification.JVLC.Sept94.pdf

> Boshernitsan and Downes, "Visual Programming Languages: A Survey", TR UCB/CSD-04-1368, December 1997.
  — http://nitsan.org/~maratb/pubs/csd-04-1368.pdf

> Burnett, "Visual Programming," Encyclopedia of Electrical and Electronics Engineering, 1999.
  — ftp://ftp.cs.orst.edu/pub/burnett/whatIsVP.pdf

> Wikipedia (!)
  — http://en.wikipedia.org/wiki/Visual_programming_language

scg.unibe.ch/scgbib?query=visprog

4

Although visual programming continues to be of current interest, curiously there are no recent survey articles available. This lecture is therefore cobbled together from a variety of sources, some of which are rather dated, but nonetheless relevant.

What is Visual Programming?

Myers (1986): "Visual Programming" (VP) refers to any system that allows the user to **specify a program in a two (or more) dimensional fashion**. Conventional textual languages are not considered two dimensional since the compiler or interpreter processes it as a long, one-dimensional stream. Visual Programming includes conventional flow charts and graphical programming languages. It does not include systems that use conventional (linear) programming languages to define pictures. This eliminates most graphics editors, like Sketchpad [Sutherland 63].

Burnett (1999): Visual programming is **programming in which more than one dimension is used to convey semantics**. Examples of such additional dimensions are the use of multidimensional objects, the use of spatial relationships, or the use of the time dimension to specify "before-after" semantic relationships.

Wikipedia (2008): A Visual programming language (VPL) is any programming language that lets users **specify programs by manipulating program elements graphically rather than by specifying them textually**. A VPL allows programming with visual expressions, spatial arrangements of text and graphic symbols. Most VPLs are based on the idea of "boxes and arrows," that is, boxes or circles or bubbles, treated as screen objects, connected by arrows, lines or arcs.

Textual programming languages are formally *one-dimensional*, since they are expressed as sequences of tokens, even though they obviously make use of two-dimensionally formatting conventions.

Visual programming inherently uses two or more dimensions, and tends to use *graphical elements* other than textual symbols.

Such rather literal definitions, however, do not reveal much insight into visual programming paradigms.
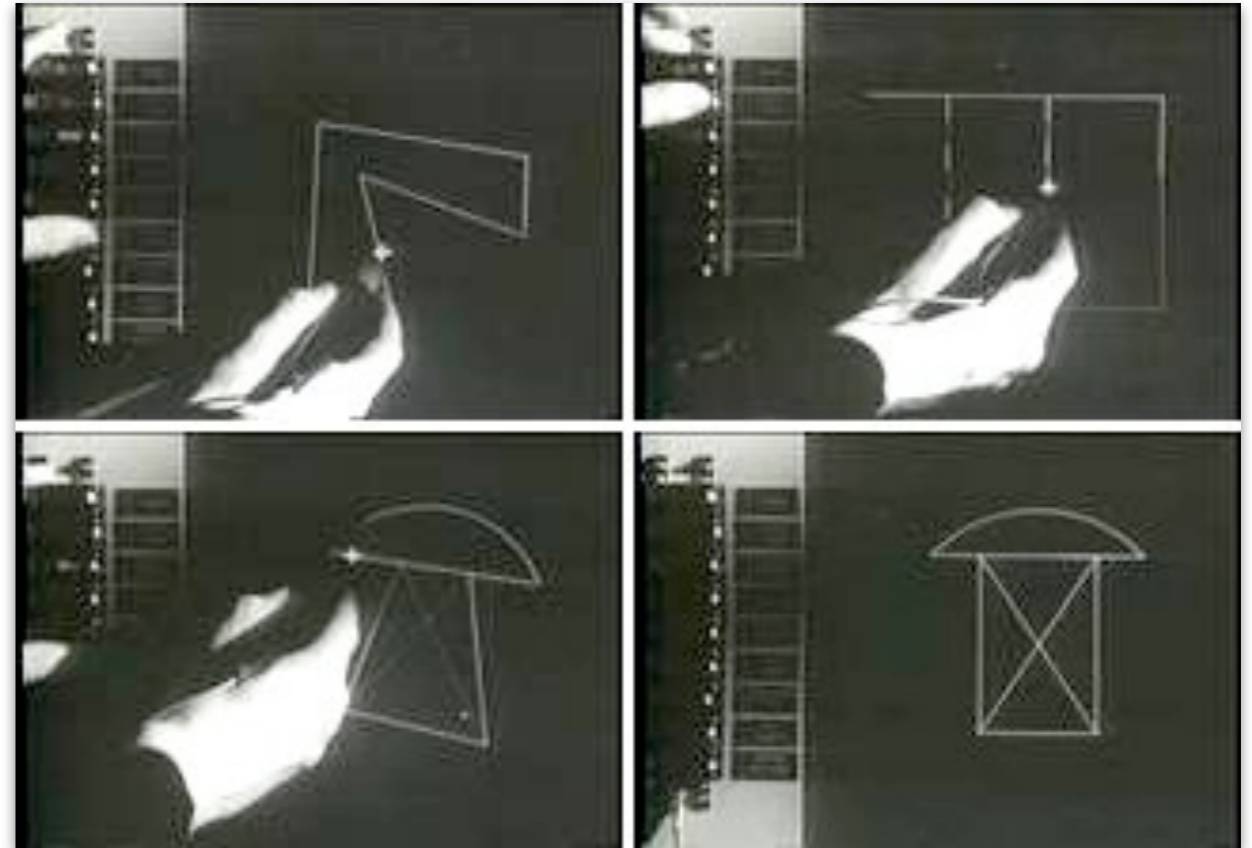
Roadmap

> Terminology
> **A Quick Tour**
> A Taxonomy of Taxonomies
> EToys (demo and evaluation)

1963: Sutherland's Sketchpad

The first computer system with a GUI, using an X-Y plotter and a light pen to construct 2D graphics.

NB: not a VPL

Ivan Edward Sutherland, *Sketchpad: A man-machine graphical communication system*, Ph.D. thesis, MIT, January 1963. www.cl.cam.ac.uk/techreports/UCAM-CL-TR-574.pdf

Sketchpad was not really a visual programming system, but rather the first computer application with a purely visual interface.

Sutherland was awarded the Turing Award in 1968 for this work.

NB: The *mouse* was also invented in 1963 by Douglas Engelbart.

See also: http://en.wikipedia.org/wiki/Sketchpad

Sutherland's PhD thesis is available online.

www.cl.cam.ac.uk/techreports/UCAM-CL-TR-574.pdf

Here is a YouTube video of the original SketchPad:

https://www.youtube.com/watch?v=6orsmFndx_o

Programming by Example
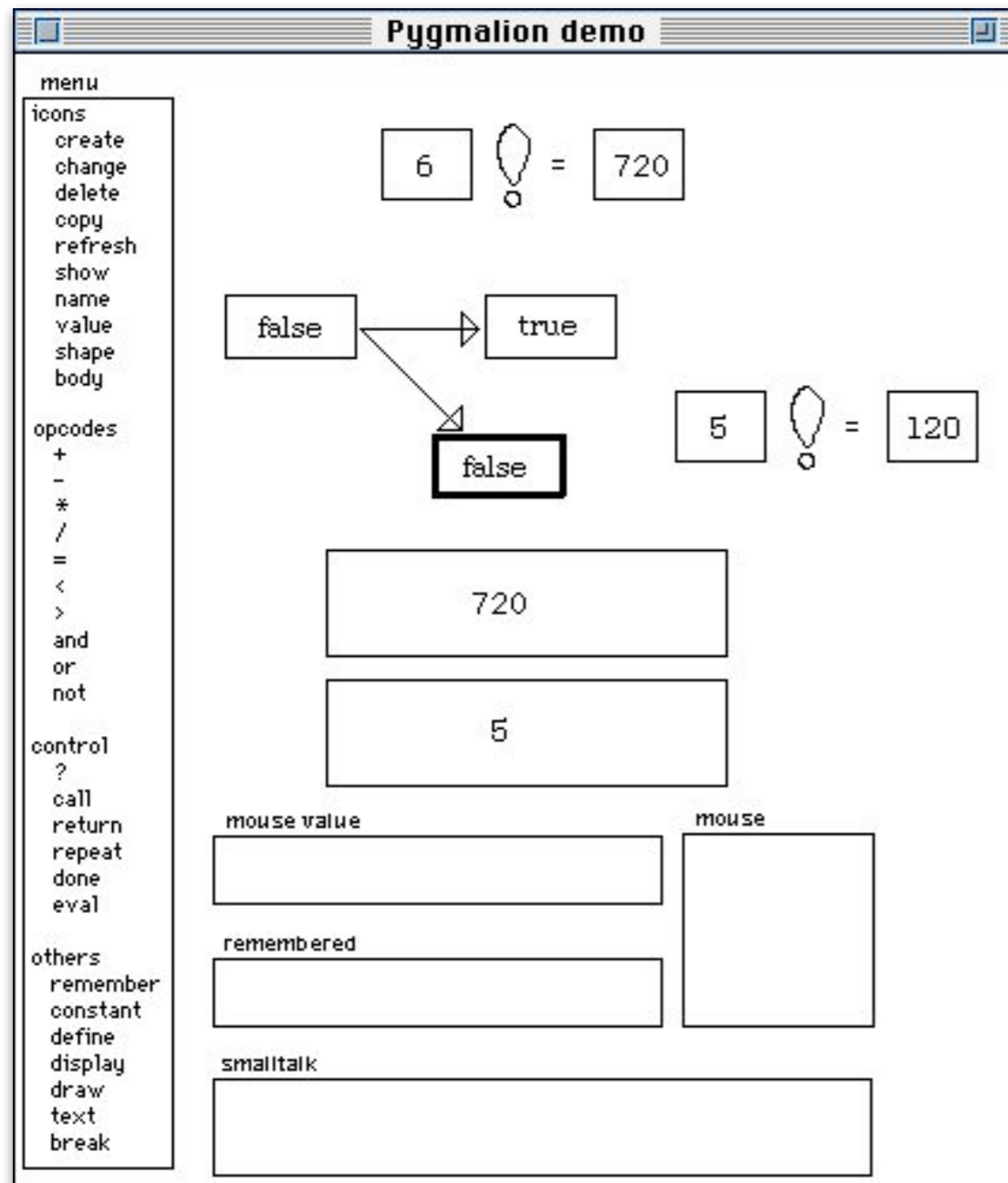
Pygmalion was an early system to let programmers interactively "demonstrate" how to compute a function. Pygmalion would then *infer* the actual algorithm.

David Canfield Smith, *"Pygmalion: a creative programming environment,"* Ph.D. thesis, Stanford University, Stanford, CA, USA, 1975.

Pygmalion introduced both the use of *icons* and the notion of *programming by demonstration*.

Here we want to "teach the computer" how to *compute a factorial*. We first *create an icon* for the function with a crude graphic. We then teach the Pygmalion how to compute the example `6!` We introduce the test `6=1`, which evaluates to false. We then instantiate a `*` function and say that we want to compute `6 * (6-1)!` (So we need a box for `6-1` and a box for `5!`) Now Pygmalion computes until it hits `1=1` which is `true`. Pygmalion asks us what to do in this case, which is to *return 1*. Finally Pygmalion blocks at the multiplication. We drag `1!` into `2 * _` and now Pygmalion knows everything and computes `6! = 720`.

Dozens of PBE systems have been implemented over the years …

Allen Cypher, et al. (Eds.), <u>Watch what I do: programming by demonstration</u>, MIT Press, Cambridge, MA, USA, 1993.

http://acypher.com/wwid/

ARK — The Alternate Reality Kit

Ark was a 2D environment for creating interactive simulations implemented in Smalltalk-80
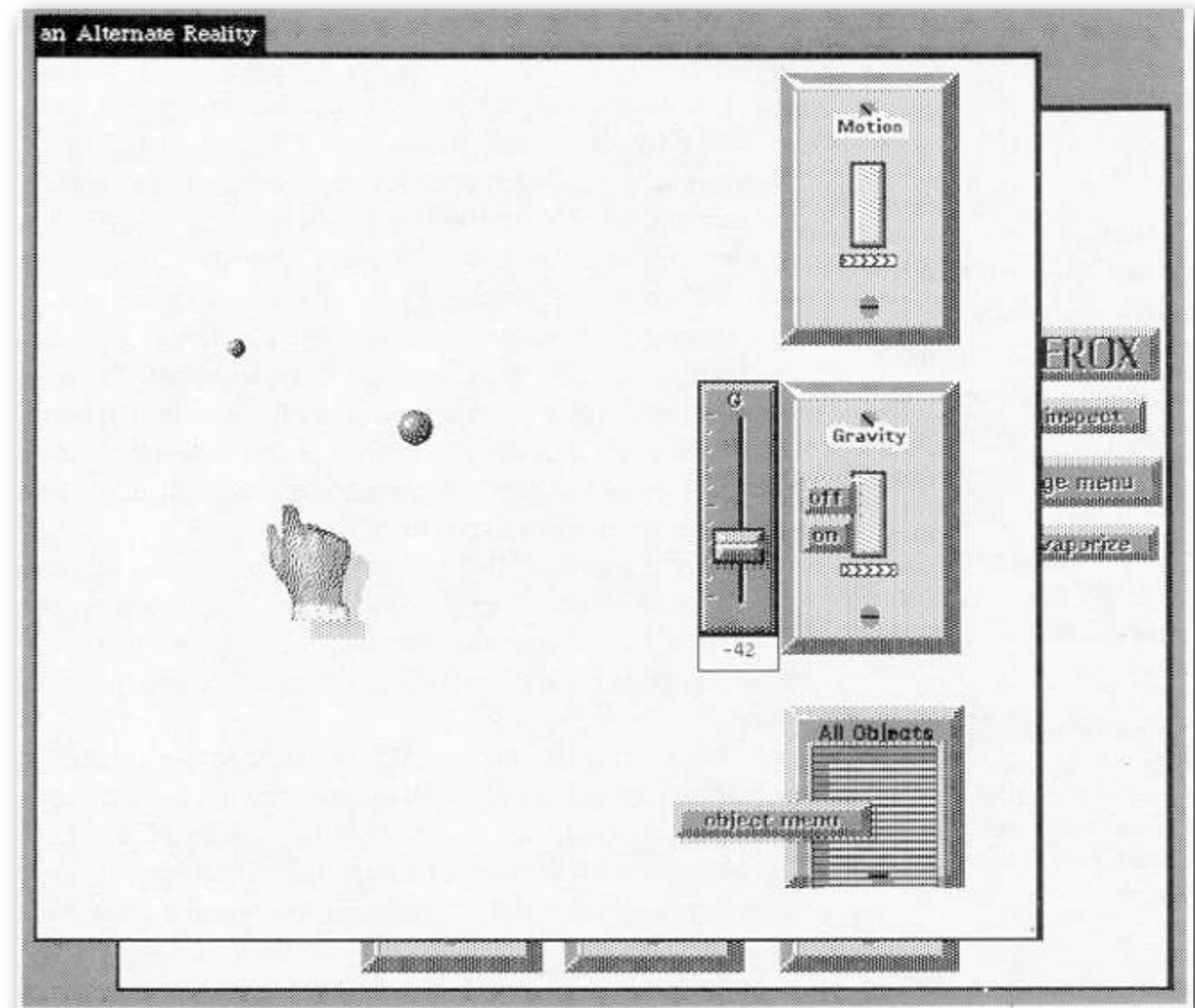
Randall B. Smith, *"Experiences with the alternate reality kit: an example of the tension between literalism and magic,"* 1987. http://dx.doi.org/10.1145/30851.30861

In ARK, objects could be created and manipulated in an environment where various physical "laws" are at work. Since then, various such physical simulation environments have been developed mainly as teaching tools for children.

ThingLab — graphical constraints

ThingLab is a graphical constraint satisfaction system implemented in Smalltalk.

Alan Borning, *Thinglab — constraint-oriented simulation laboratory*, Ph.D. thesis, Stanford University, Stanford, CA, USA, 1979. www.2share.com/thinglab/ThingLab%20-%20index.html

ThingLab is another simulation environment, where the computational paradigm is that of *constraint satisfaction.*

In this example, two bar widgets are connected using (bi-directional)) constraints that convert between Celsius and Fahrenheit temperatures. If you adjust the temperature in either widget, the other one will automatically adjust.

For details, see the ACM TOPLAS paper:

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.2858&rep=rep1&type=pdf

Fabrik — bidirectional dataflow

With Fabrik, you could build computations and GUIs using bidirectional dataflow instead of constraints.
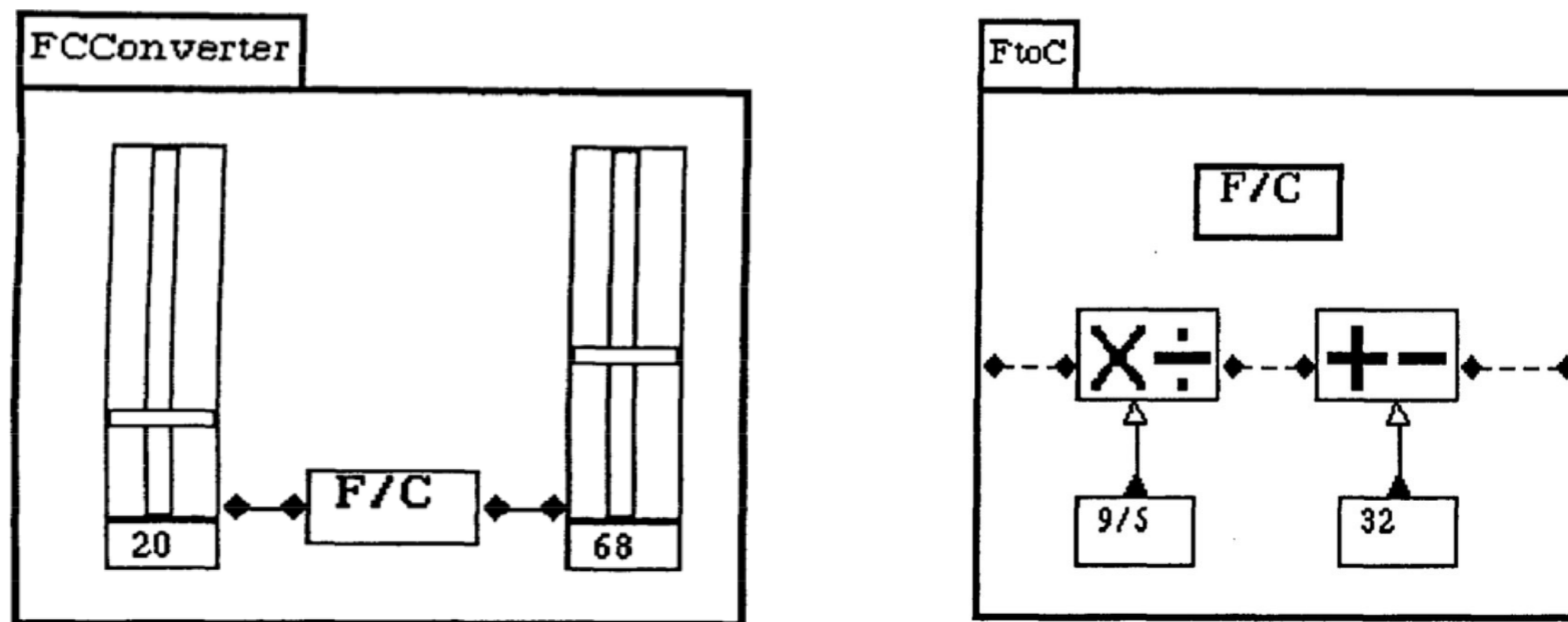
Fabrik adopted a *components-and-connectors* approach to general purpose visual programming. Instead of using constraints, Fabrik was based on dataflow. The Fahrenheit-Celsius converter uses two slider components connected with a bi-directional dataflow component that performs the calculation.

Architectural Description Languages

Architectural Description Languages (ADLs) model systems in terms of
- *components* that offer services,
- *connectors* that bind services, and
- architectural *constraints* that must be respected.

As a consequence, certain system *properties* are obtained.
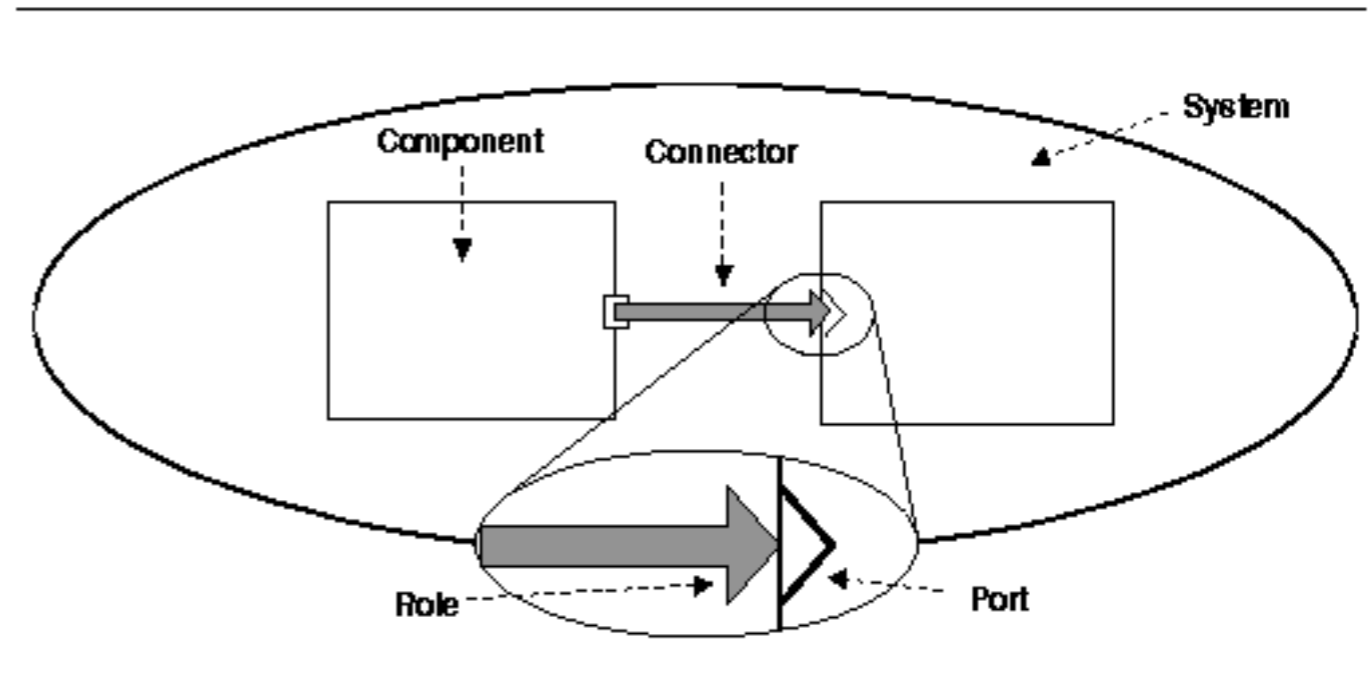
www.cs.cmu.edu/~acme/docs/language_overview.html

Mary Shaw and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

ADLs express (software) architectural constraints structurally in terms of *components* and *connectors*. The constraints are intended to guarantee certain desirable properties. For example, in a layered design, the components are layers, and the connectors are the mono-directional APIs offered by a given layer to the layer above. The constraint is that a layer may only address the layer below. The property guaranteed is that changes within a layer do not affect other layers; changes to an API only affect the layer above.

Although ADLs are often expressed as *visual languages*, only some are visual *programming* languages. Generally these are components and connectors builders.

Components and Connectors

Vista — a Visual Scripting Language
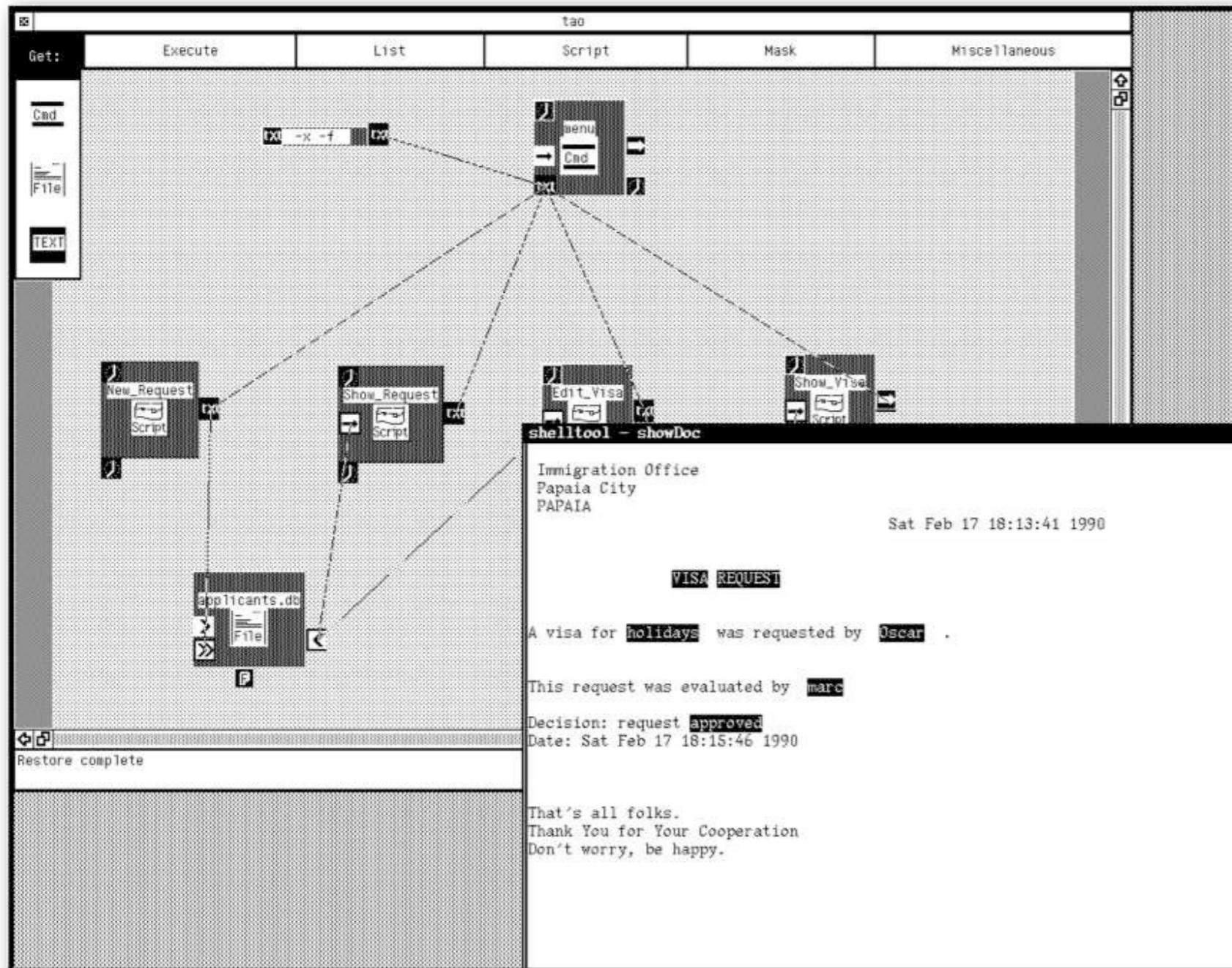
Many ADLs provide a components and connectors graphical tool interface:
- ConicDraw (Imperial College)
- Vista (U Geneva)
- Wright (CMU)
- ACME (CMU/USC)

Most components-and-connectors tools allow you to connect input and output ports of components to build an application.

The semantics of connections vary — sometimes it is dataflow, but more generally a connection stands for binding required and provided services.

Some tools allow composite components to be built from parts.

The screenshot is from Vista, a tool developed at the University of Geneva in the early 90s. The underlying components were Unix programs.

Prograph — dataflow graphs

Prograph is a visual, object-oriented dataflow language.

A product during the 1990s. Now reborn as Andescotia "Marten" for Mac OSX. (www.andescotia.com)
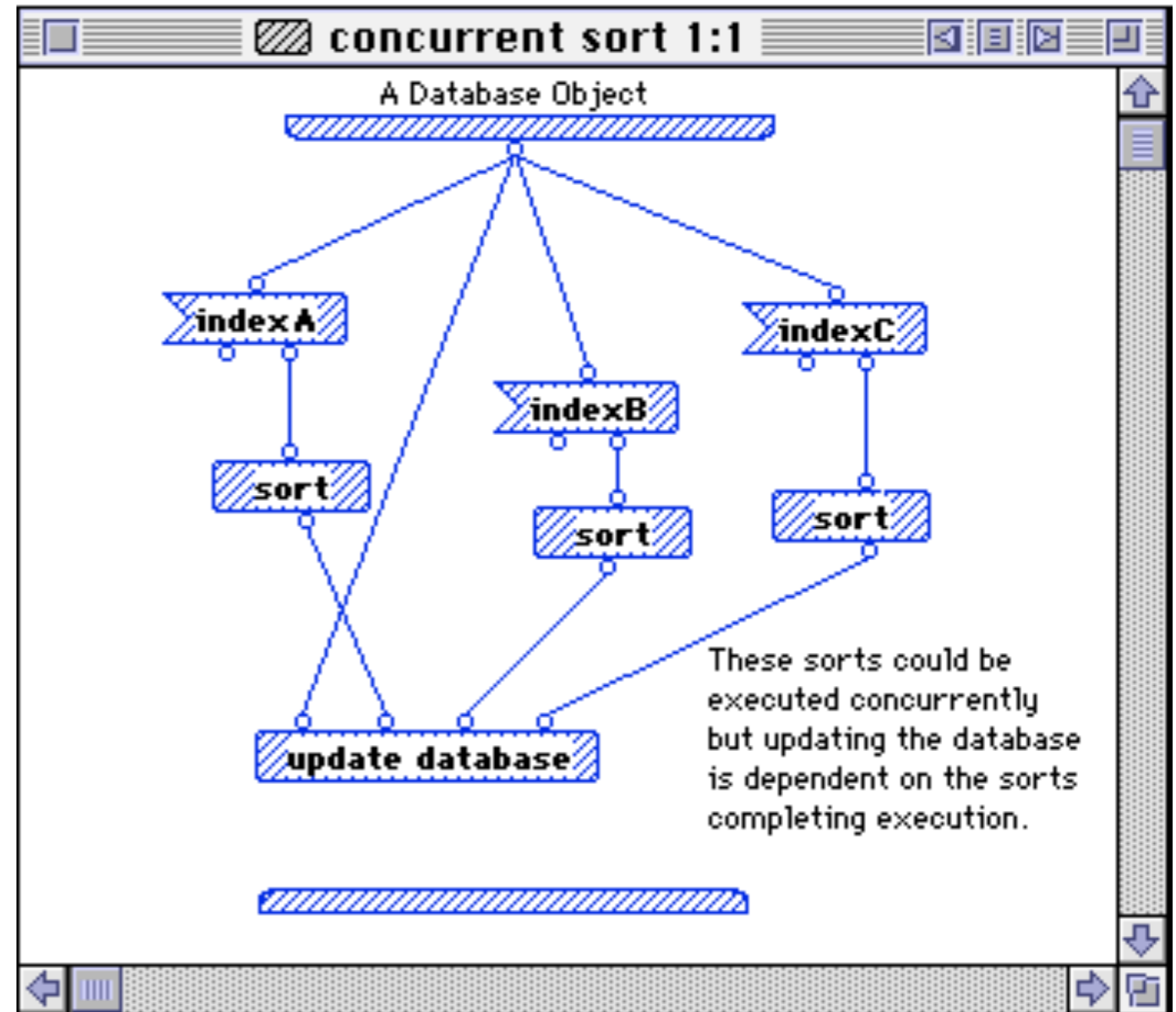
A *Prograph* program is a *directed graph* of connected dataflow components.

Paragraph supports composite components, iteration with failure handling, and various other control constructs.

There are 10 basic data types (Boolean, integer, list, object …) and 307 (!) primitives.

Research started in 1982 as part of a course at Acadia University on functional and dataflow languages.

A prototype was developed 1983-1985, then work started on a commercial tool.

The tool was called Prograph from 1990-1995. A new company Pictorius then formed.

https://en.wikipedia.org/wiki/Prograph

Yahoo Pipes — mashup dataflow

Mashup internet resources by composing pipes and filters

Yahoo pipes used the components-and-connectors paradigm to allow users to visually compose mashups.

This data flow script filters out the blog articles from the PhD comics feed and only shows the comics.

https://en.wikipedia.org/wiki/Yahoo!_Pipes

EToys — Tile-Based Programming

Program
simulations
by composing
"tiles"

EToys uses the *tile-based programming* paradigm. Graphical elements are controlled by scripts that are composed of jigsaw-like *tiles* that can only be combined in a fixed manner. The only text you type are the names of objects, scripts and variables.

Everything else is done by dragging and dropping tiles.

Scripts manipulate graphical objects (Morphs) which can interact with each other and with the user and the environment.

EToys can be downloaded from SqueakLand.

More on this later.

https://en.wikipedia.org/wiki/Etoys_(programming_language)

http://www.squeakland.org

OSX Automator — Workflow-based scripting

OSX Automator is a built-in tool for scripting common actions as "workflows".
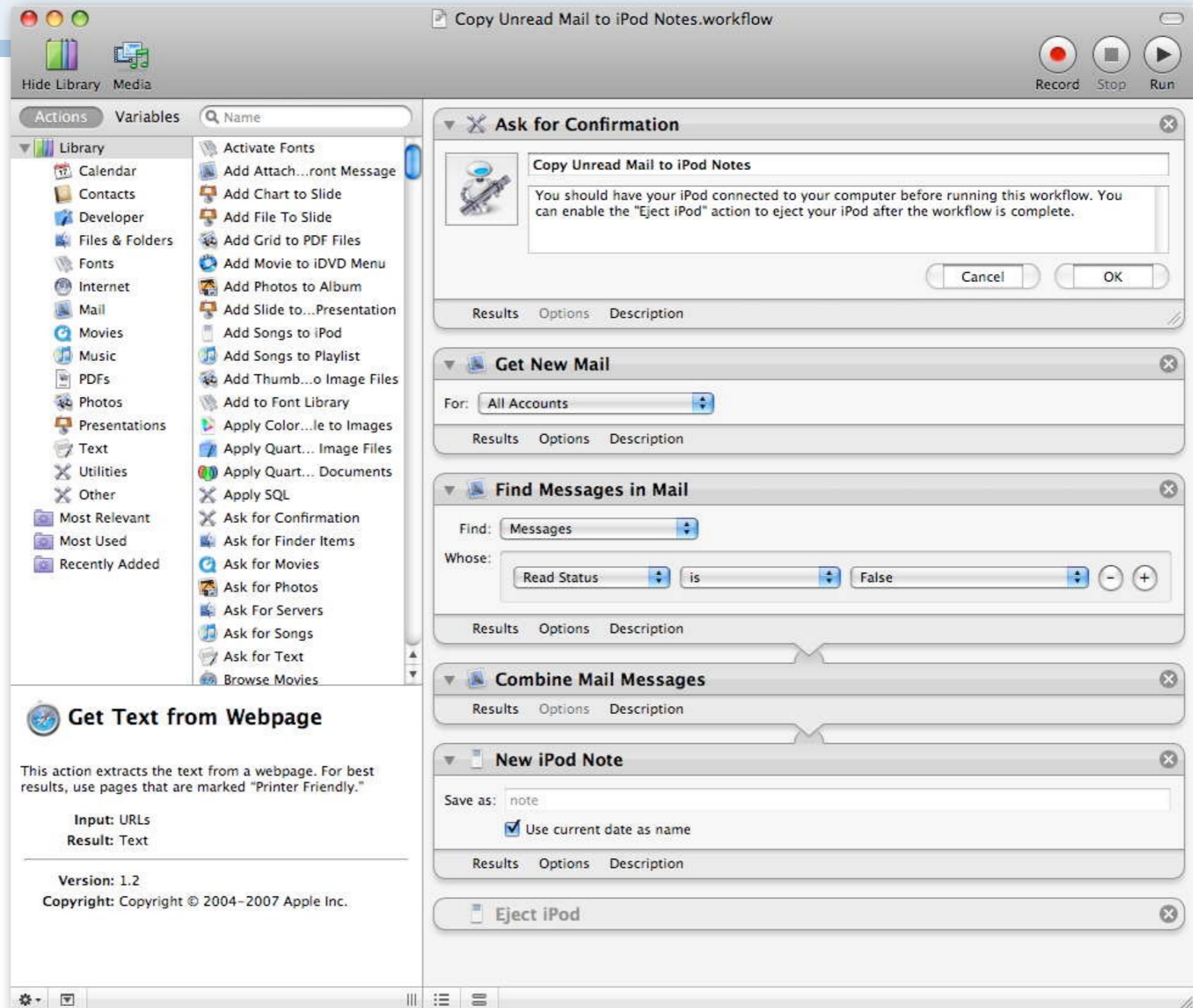
Naked Objects — visual domain objects

Instead of coding all
4 tiers by hand …
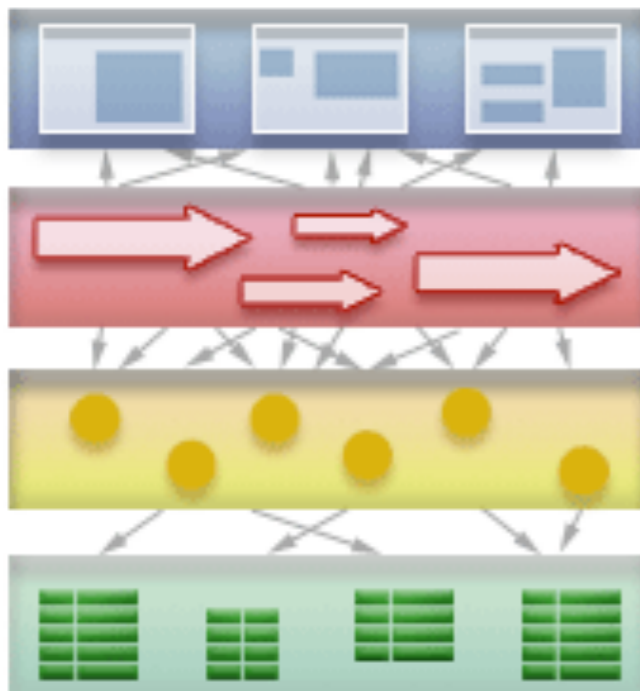
… generate both
persistence layer
and UI automatically
from domain objects.
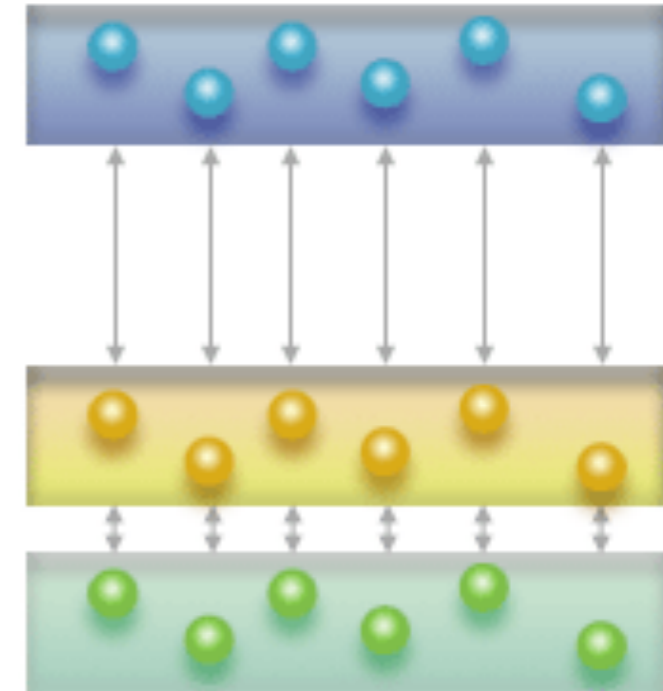
www.nakedobjects.org
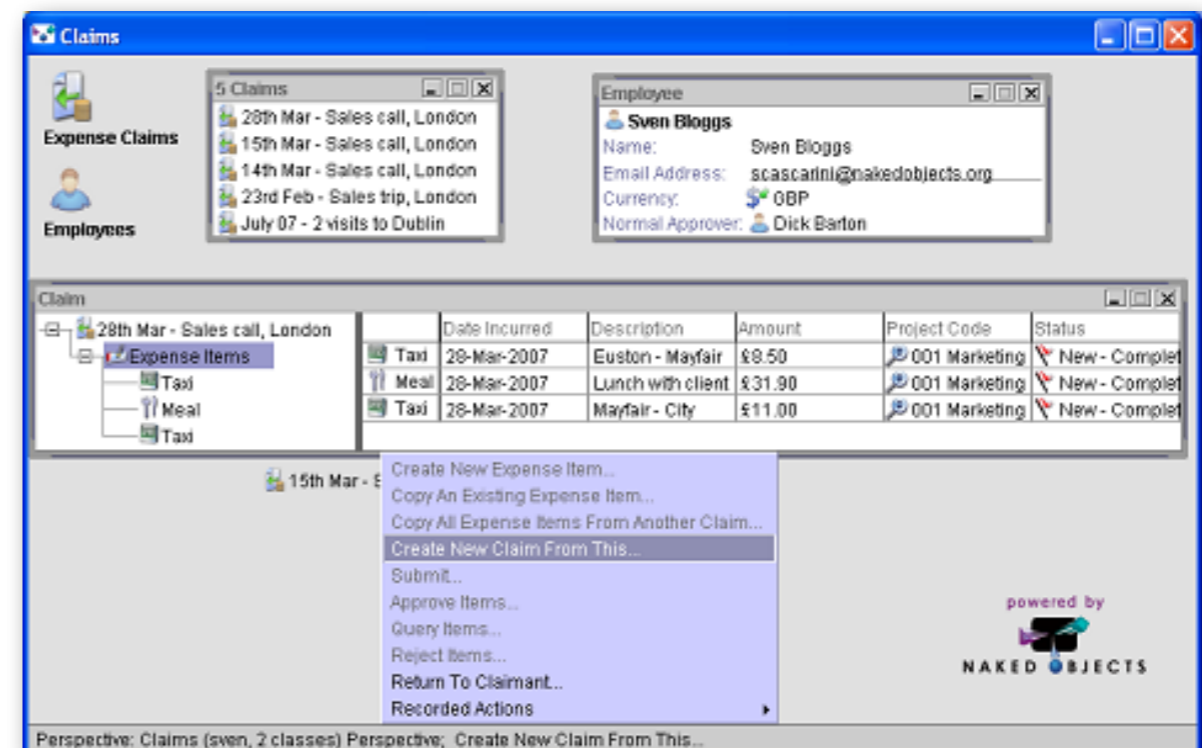
"*Naked Objects*" is both an *architectural pattern* and an open source project.

See also *Pawson's PhD thesis*.

Naked Objects is often contrasted to MVC, but actually closer in spirit to the original idea. (See Reenskaug's preface to the thesis.)

https://en.wikipedia.org/wiki/Naked_objects

http://downloads.nakedobjects.net/resources/Pawson%20thesis.pdf

Subtext —example-centric schematic tables
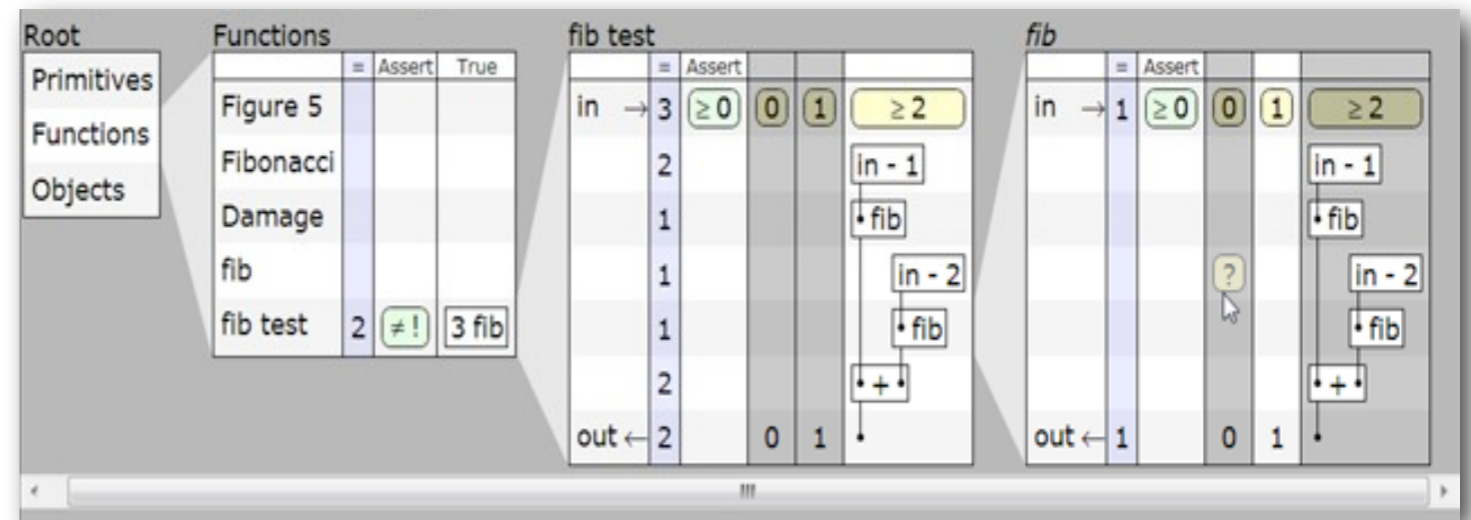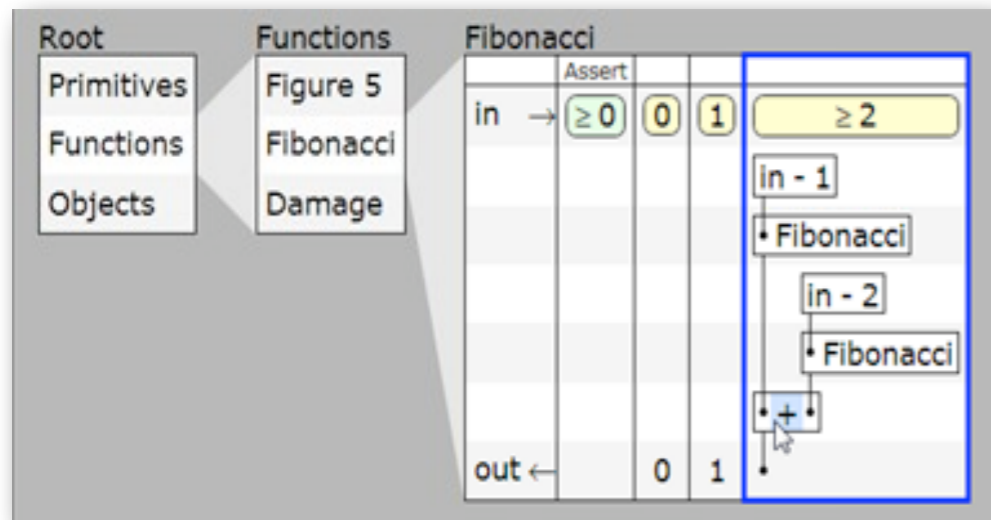
The Central Idea
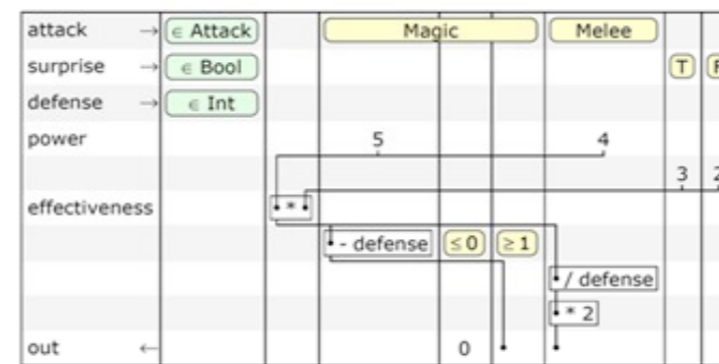
DOING
Computation
Semantics:
function graph
Presentation:
spanning tree

DECIDING
Logic
Semantics: Boolean Algebra
Presentation: partitioned columns

Jonathan Edwards, "*No ifs, ands, or buts: uncovering the simplicity of conditionals*," *OOPSLA* 2007. http://dx.doi.org/10.1145/1297027.1297075

http://subtextual.org/

Subtext encodes program logic as *visual tables* whose *columns represent logical alternatives*, and *rows represent computational elements*. Interestingly, the same tables can represent both conventional control structures (if-then-else) as well as polymorphic dispatch. (Logical choices can represent subclasses.)

Some affinity to spreadsheets is claimed.

The tool is example-centric — as you edit, the tables compute.

There is a very nice video demonstration on the web site. No public download available however.

  https://en.wikipedia.org/wiki/Subtext_(programming_language)

AGG — graph transformation rules

"*AGG* is a rule based visual language supporting an algebraic approach to graph transformation."

http://tfs.cs.tu-berlin.de/agg/
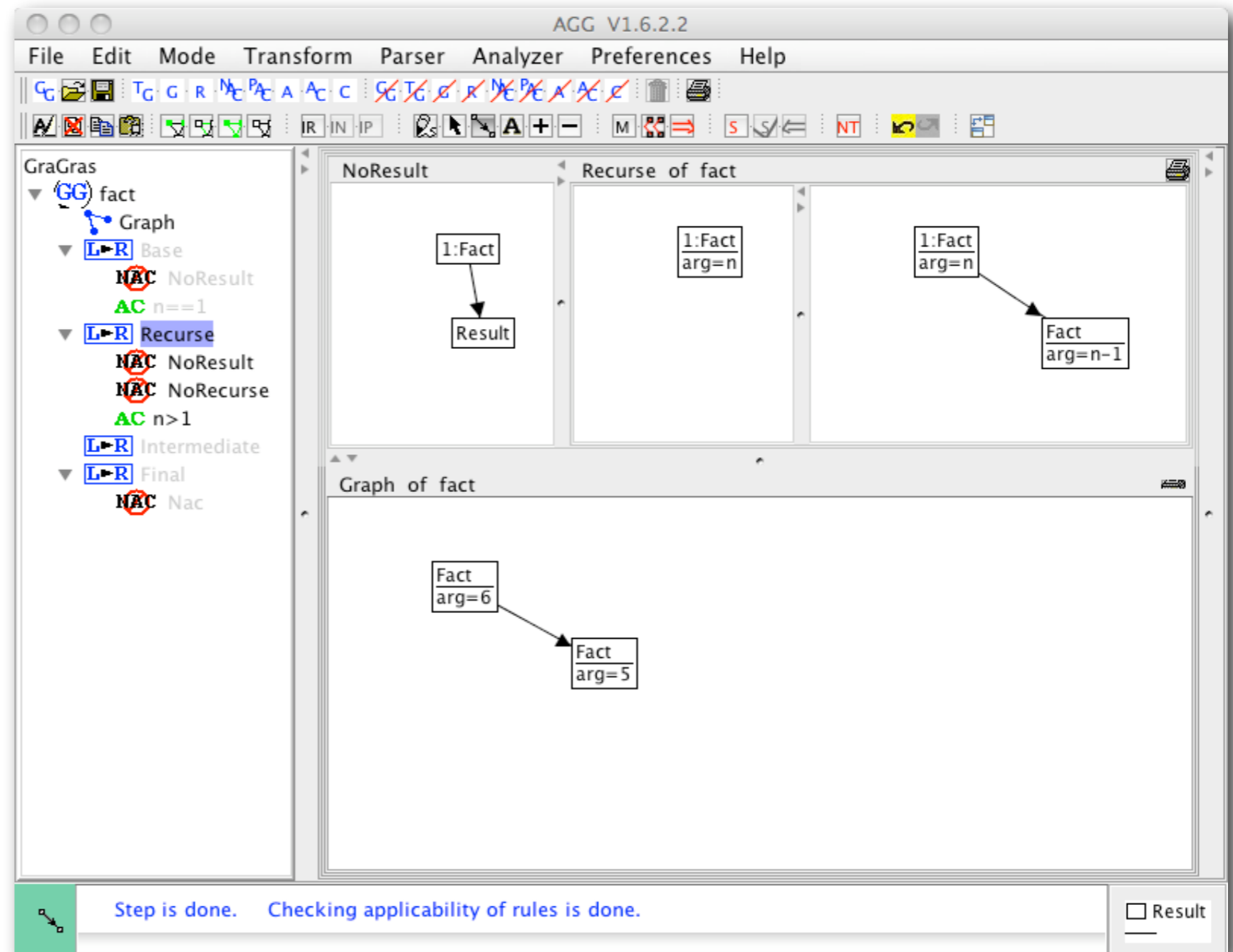
AGG (Attributed Graph Grammar System) is an evolution of *graph grammars*. Basically you specify rules that transform parts of graphs to new graphs. A key application is as a tool to support UML metamodel transformations.

This demo is in the example repo and is inspired by Pygmalion. The graph starts with a factorial node that has an argument attribute set. The rules will transform this to a final `Result` node.

Here we see the graph after one step of the `Recurse` rule having been applied. This rule transforms a `Fact` node to add a new subnode to compute the recursive factorial. There are two negative application conditions (NACs: no result or recursive factorial exists already), and one attribute condition (AC: n>1).

There is a canned video available of the demo:

http://scg.unibe.ch/download/Demos/Videos/AGG-demo.mp4

A number of similar graph transformation tools exist.

https://www.cs.le.ac.uk/people/rh122/gratra/applications.html

Roadmap

> Terminology
> A Quick Tour
> **A Taxonomy of Taxonomies**
> EToys (demo and evaluation)

Meyers, 1986 — a 2^3 partition

Not Programming by Example

	Batch	Interactive
Not VP	4.1 All Conventional Languages: Pascal, Fortran, etc.	4.1 LISP, APL, etc.

VP	4.2
	
	Grail
	  [Ellis 69]
	AMBIT/G/L
	  [Christensen 68,71]
	Query by Example
	  [Zloof 77, 81]
	FORMAL
	  [Shu 85]
	GAL
	  [Albizuri-Romero 84]
		4.3
		
		Graphical Program Editor
		  [Sutherland 66]
		PIGS
		  [Pong 83]
		Pict
		  [Glinert 84]
		PROGRAPH
		  [Pietrzykowski 83,84]
		State Transition UIMS
		  [Jacob 85]

Programming by Example

	Batch	Interactive

Not VP	4.4
	
	I/O pairs*
	  [Shaw 75]
		4.5
		
		Tinker
		  [Lieberman 82]

VP	4.6
	
	[Bauer 78] traces*
		4.7
		
		AutoProgrammer*
		  [Biermann 76b]
		Pygmalion
		  [Smith 77]
		Graphical Thinglab
		  [Borning 86]
		SmallStar
		  [Halbert 81,84]
		Rehearsal World
		  [Gould 84]

Figure 1.

Classification of programming systems by whether they are visual or not, whether they have Programming by Example or not, and whether they are interactive or batch. The small numbers refer to the section in which the group is discussed. Starred systems (*) have inferencing, and non-starred PBE systems use Programming With Example.

This rather old taxonomy includes many things that are not VPLs, and does not really offer any detailed insight into the design space of VPLs.

Basically the lower right quadrants are the real VPLs (VP+Interactive).
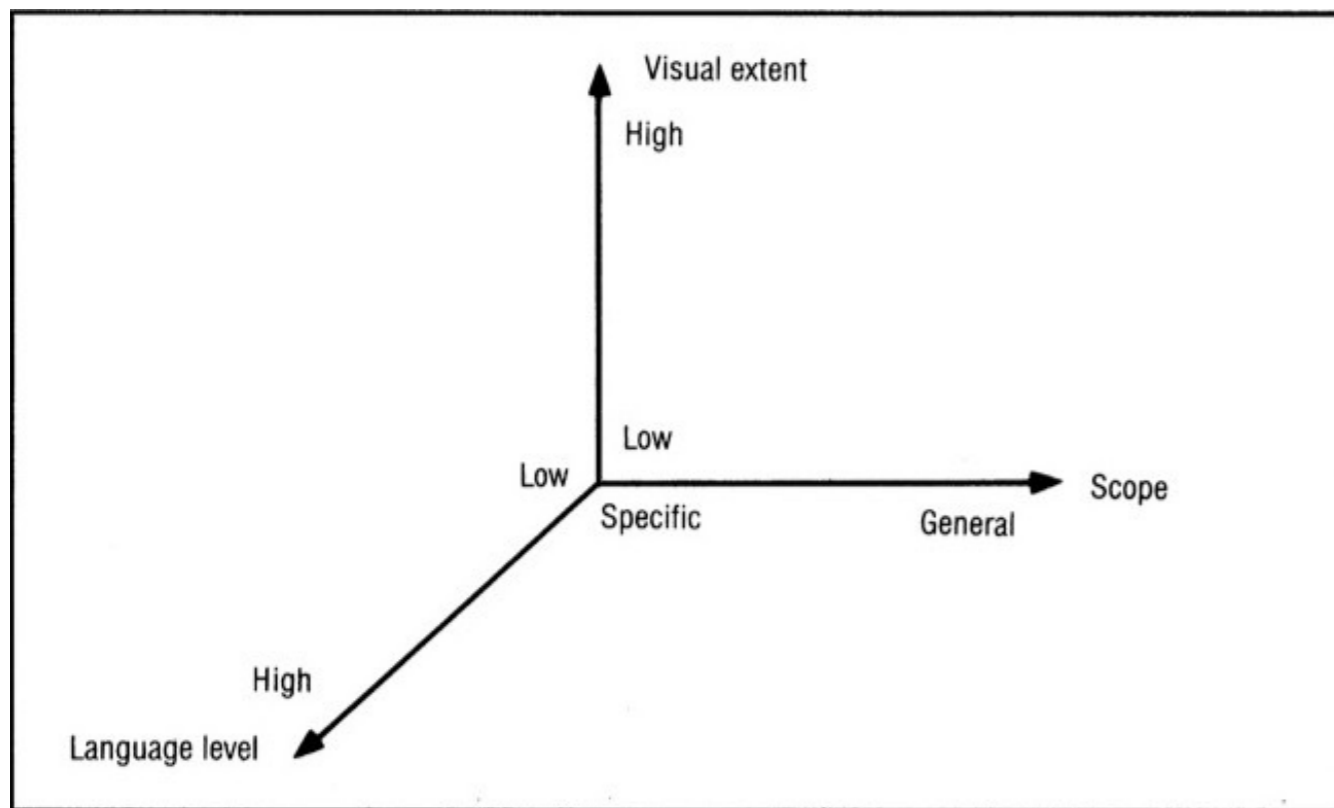
Chang/Shu, 1987 — a 3 dimensional scale

Figure 1. A three-dimensional framework to characterize and compare visual languages.

In Shu's excellent introduction and survey of visual languages, she proposes a three-dimensional framework to characterize and compare visual languages. To evaluate if the visual language approach is adequate for an intended application with a certain type of user, three questions should be asked:

(1) Is it adequate for visualization?
(2) Is it adequate for representing processes?
(3) Is it adequate for representing objects?

Programming by rehearsal. The Programming-by-Rehearsal system[7] is a visual programming environment implemented in Smalltalk-80 on the Xerox Lisp Machine. It provides a powerful metaphor for visual programming. A rehearsal world is created by (1) and...

...in a troupe in the display, as Figure This system is high in visual content, but low in level and scope because only icons on the screen can be manipulated. However, the program design process is quick, easy, and enjoyable; a simple program can be created in less than 30 minutes.

Shi-Kuo Chang, "*Visual languages: a tutorial and survey*," IEEE Software, 1987
http://dx.doi.org/10.1109/MS.1987.229792

23

This survey of visual languages focuses on visualization, not programming.

Burnett, 1994 — an empirical classification for research papers

Table 1. The Visual Programming Language classification system.

VPL: Visual Programming Languages
VPL-I. Environments and Tools for VPLs
VPL-II. Language Classifications
  A. Paradigms
    1. Concurrent languages
    2. Constraint-based languages
    3. Data-flow languages
    4. Form-based and spreadsheed-based languages
    5. Functional languages
    6. Imperative languages
    7. Logic languages
    8. Multi-paradigm languages
    9. Object-oriented languages
    10. Programming-by-demonstration languages
    11. Rule-based languages
  B. Visual representations
    1. Diagrammatic languages
    2. Iconic languages
    3. Languages based on static pictorial sequences
VPL-III. Language Features
  A. Abstraction
    1. Data abstraction
    2. Procedural abstraction
  B. Control flow
  C. Data types and structures
  D. Documentation
  E. Event handling
  F. Exception handling

VPL-IV. Language Implementation Issues
  A. Computational approaches (e.g. demand-driven, data-driven)
  B. Efficiency
  C. Parsing
  D. Translators (interpreters and compilers)
VPL-V. Language Purpose
  A. General-purpose languages
  B. Database languages
  C. Image-processing languages
  D. Scientific visualization languages
  E. User-interface generation languages
VPL-VI. Theory of VPLs
  A. Formal definition of VPLs
  B. Icon theory
  C. Language design issues
    1. Cognitive and user-interface design issues (e.g. usability studies, graphical perception)
    2. Effective use of screen real estate
    3. Liveness
    4. Scope
    5. Type checking and type theory
    6. Visual representation issues (e.g. static representation, animation)

24

This paper offers a taxonomy for classifying research papers, not VPLs. VPL-II focuses on language paradigms.

The classification has been empirically tested on a set of research papers.

Time for a new taxonomy?

1. What are the *visual elements*?
   - > Icons, graphs, tables, forms …
   - > How much text? Purely visual, or mixed?

2. What *paradigm* is used?
   - > PBE, constraints, dataflow, tile composition, components and connectors, graph transformation …

3. What is the *application domain*?
   - > Simulation, games, animations, modeling, component composition, algorithms …

4. What is the *target audience*?
   - > Beginner? Domain specialist?

Astonishingly there exists no proper, up-to-date survey of VPLs today.

The dimensions listed here reflect the results of a seminar project that surveyed over 100 past and present VPLs.

This is not a full-fledged taxonomy, but just a sketch of some of the design dimensions for VPLs. Clearly the dimensions are not orthogonal, and there can be many overlaps between the various criteria identified.

Roadmap

> Terminology
> A Quick Tour
> A Taxonomy of Taxonomies
> **EToys (demo and evaluation)**

EToys references

> Allen-Conn and Rose, *Powerful Ideas in the Classroom*, Viewpoints Research Institute, Inc., 2003.

  — www.squeakland.org/sqmedia/books/order.html

> Gaelli, *Composing Simple Games with EToys*

  — www.emergent.de/etoys.html

> Gaelli, et al., *Idioms for Composing Games with EToys*, C5 2006

  — scg.unibe.ch/archive/papers/Gael06aC5.pdf

EToys in a nutshell

> **"The GUI is the model"**
  — No MVC — "morphs" are graphical objects with behaviour
  — Prototype-based — morphic framework ported to Squeak from Self

> **Tile-based programming**
  — The only thing you type are names: Scripts, Objects and Variables
  — The rest is composed via drag and drop of tiles.

> **Toolbox of existing objects**
  — Numerous pre-packaged morphs are available with special behaviour

> **Build your own**
  — You can compose your own morphs from the tookit
  — Or you can program new kinds of Morphs in Smalltalk

Squeakland — Squeak for Educators

Squeakland's version of Squeak supports a plugin to run Squeak "projects" directly in your browser.

29

Squeak Projects

Squeak <u>projects</u> can be used to save the state of a set of *objects*, (i.e, not just source code).

A project

"Publish" a project (i.e., as a file that can be loaded from Squeak or from a browser plugin)
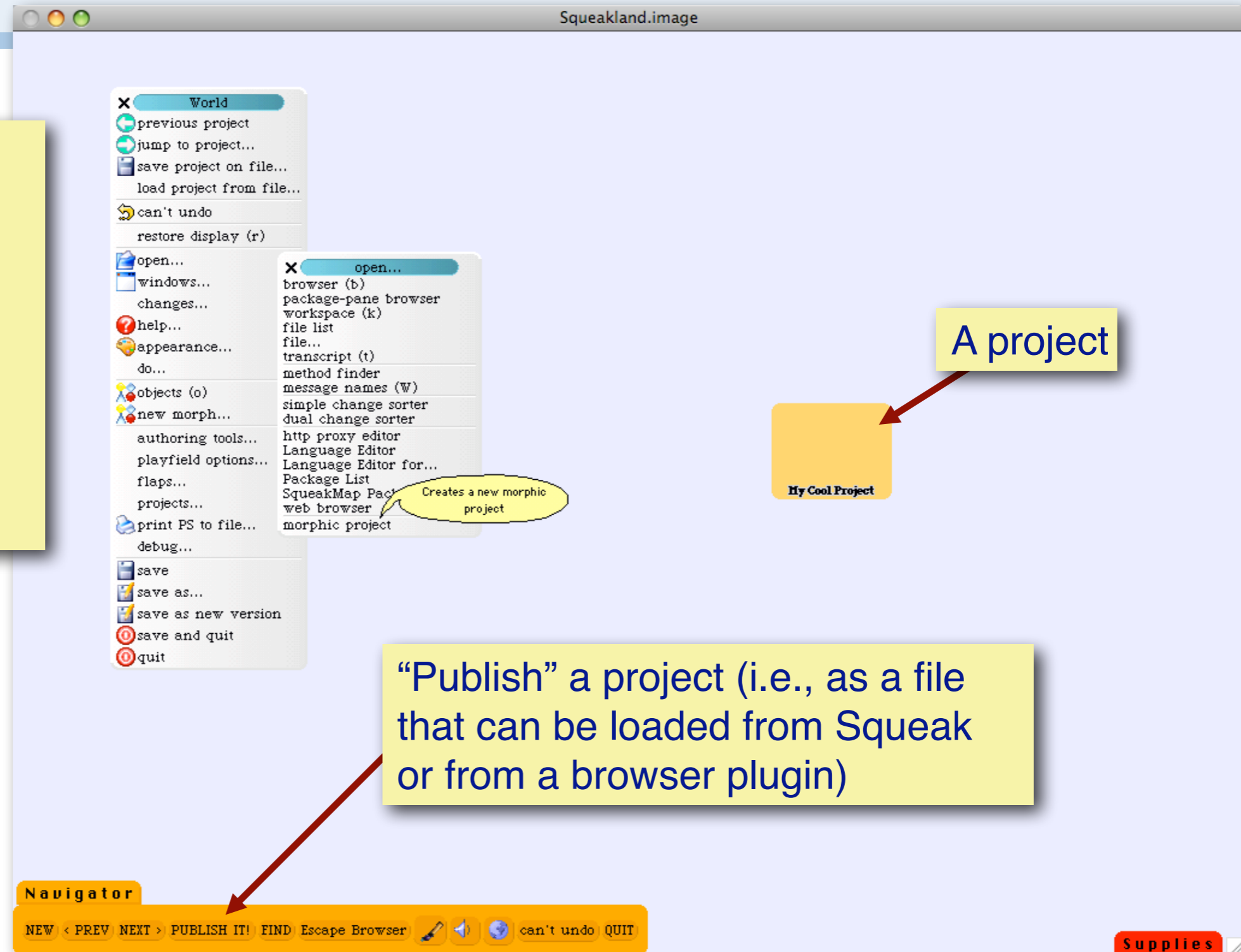
The eToys demos are available here:

```
git clone git@scg.unibe.ch:lectures-pa-examples
```

The demos are all canned projects that can be loaded into the Squeakland eToys image.

Morphic objects in Squeak

Object catalog
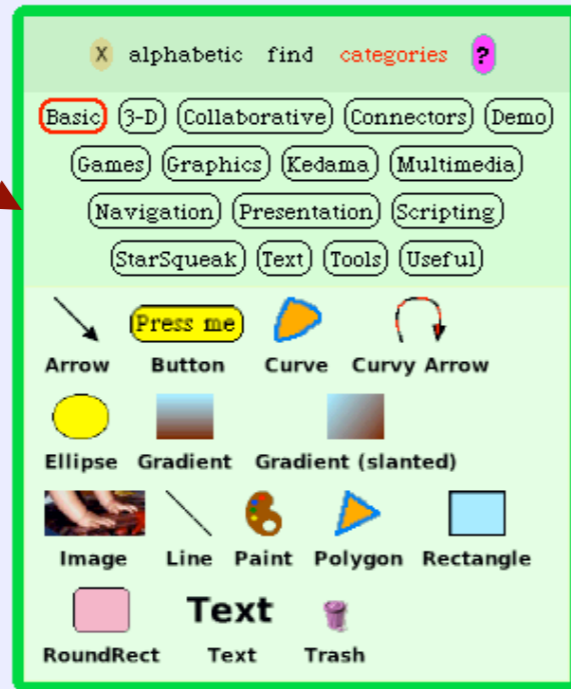
A morph with "handles"
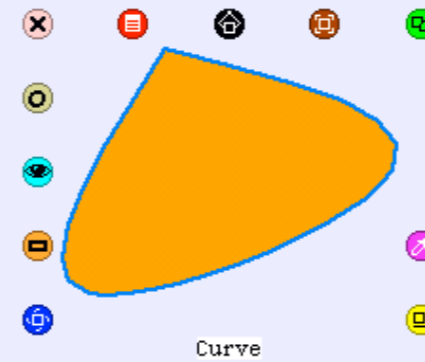
A flap with various supplies

31
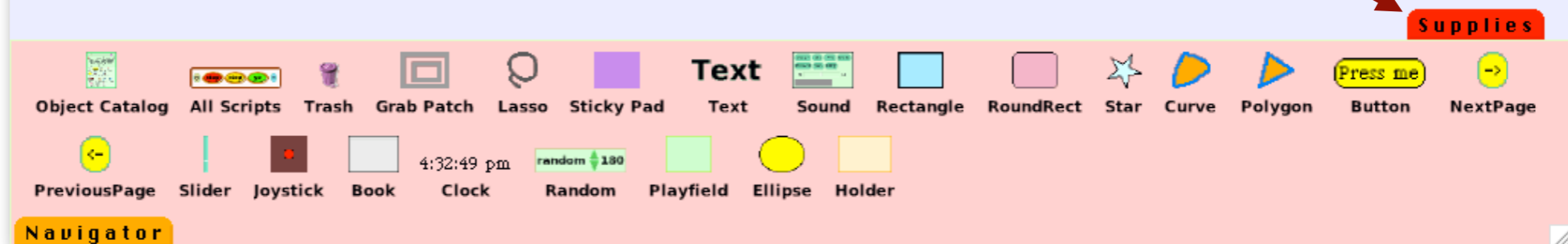
Morphic is the graphical framework originally developed by Randall Smith and John Maloney for Self, a prototype-based language inspired by Smalltalk.

Morphic was ported to Squeak and served as the foundation for eToys.

A morphic object can be selected with an "option-click" to reveal its "morphic handles". These consist of a graphical menu of operators that allow you to rotate it, clone it, debug it, and so on.

The "object catalog" and the "supplies flap" provide a number of pre-defined morphic objects.

The canonical car demo — step 1: paint a car
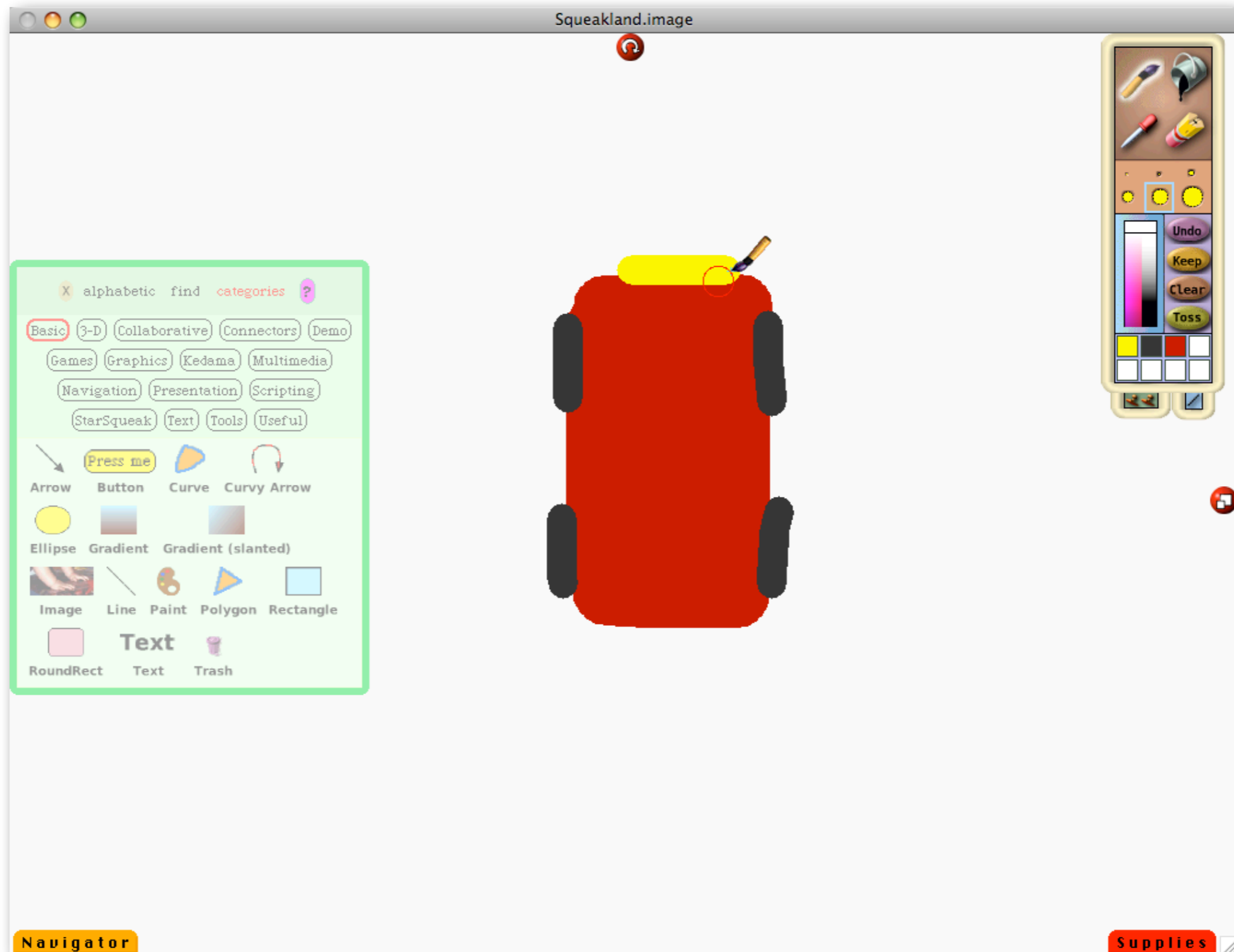
This is the canonical eToys demo. We draw a simple graphic of a car, and script it so it races around a drawing of a race track.

A canned video is also available from the SCG web site.

http://scg.unibe.ch/download/Demos/Videos/eToys-car-demo.mp4
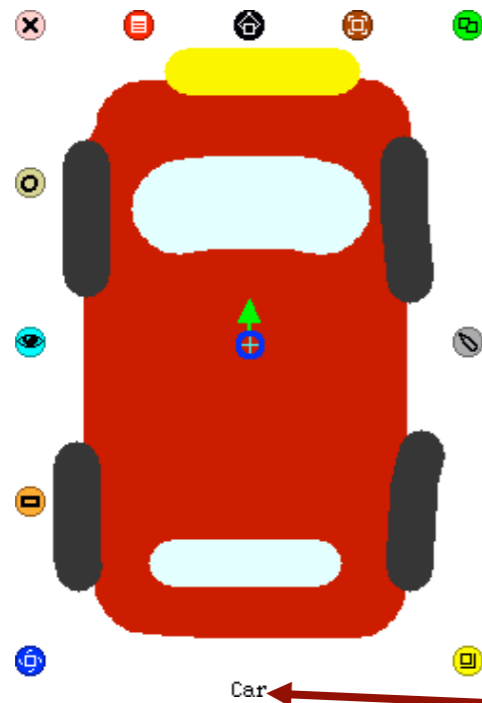
Morphic "handles"

Morphic handles are used to manipulate and script graphical objects

Click on the name to change it

⊙ **Collapse** - Hides your object.

▣ **Copy** - Makes a copy of your object.

▤ **Menu** - Contains useful tools for objects.

▣ **Move** - Lets you move your object.

⬡ **Pick Up** - Lifts your object out of its container.

✎ **Repaint** - Lets you repaint your object.

▣ **Resize** - Lets you resize your object.

⊕ **Rotate** - Lets you rotate your object.

▬ **Tile** - Brings up a tile with your object's name.

✕ **Trash** - Moves your object to the trash.

👁 **Viewer** - Opens a viewer for your object.

33

EToys — scripting objects by composing *tiles*

Open an object's Viewer to see the commands it understands.

Commands can be dragged out to form *tiles*, which can be composed to *script* new commands

Squeak Scripting

Edit name
Remove viewer from screen
Remove pane
Search for methods
Previous/Next category
Run command once
New menu
Drag from here to get an assignment phrase

Viewer

The Scriptee
Script name
When the script runs
Test for yes or no
Script menu

Script

Control Buttons

View all scripts
Run all scripts
Run paused scripts once
Pause ticking scripts
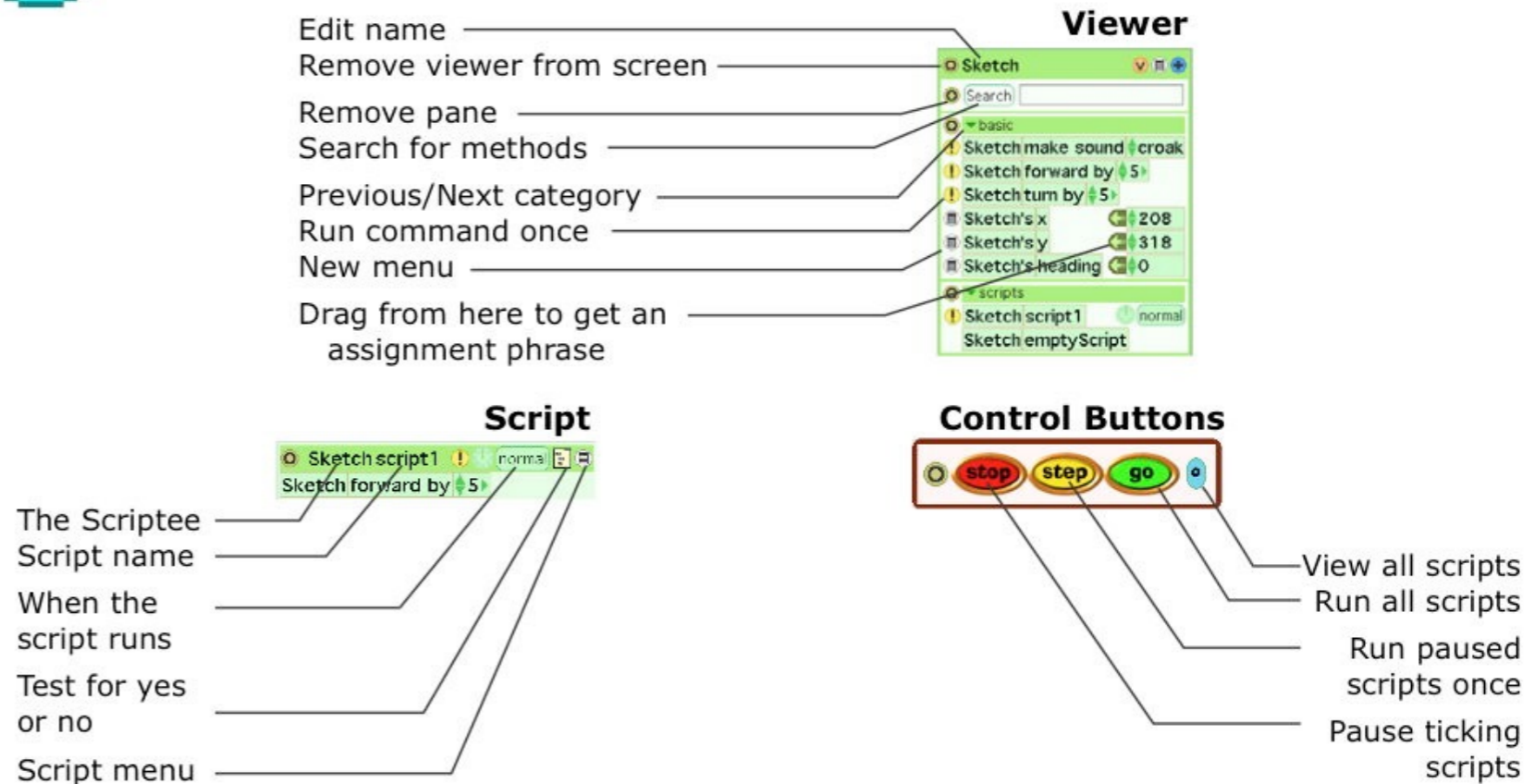
Step 2: make the car move

Click to run this command

Step 3: drag out commands to compose a script
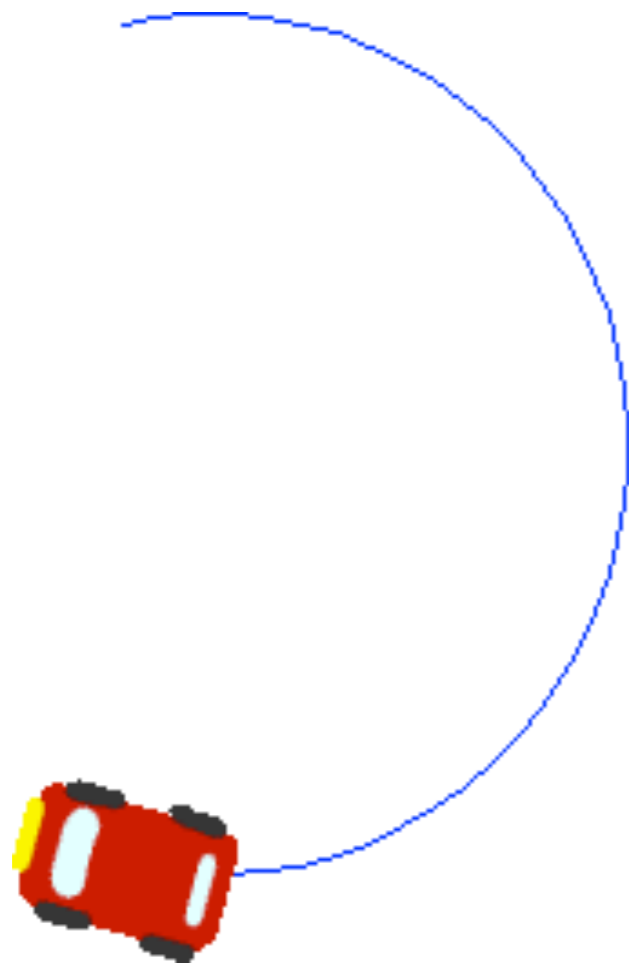
Step 4: paint a racing track
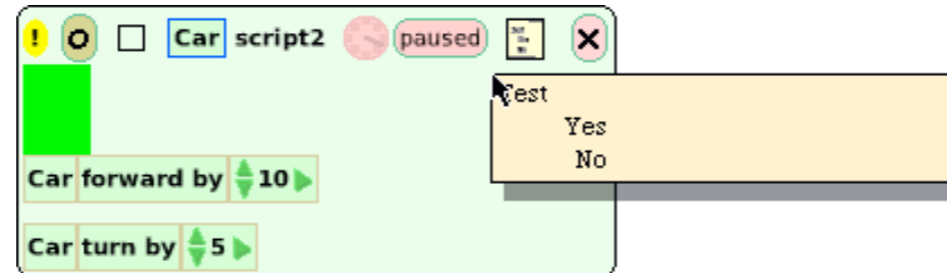
We now add a conditional to make the car turn if it veers off the track
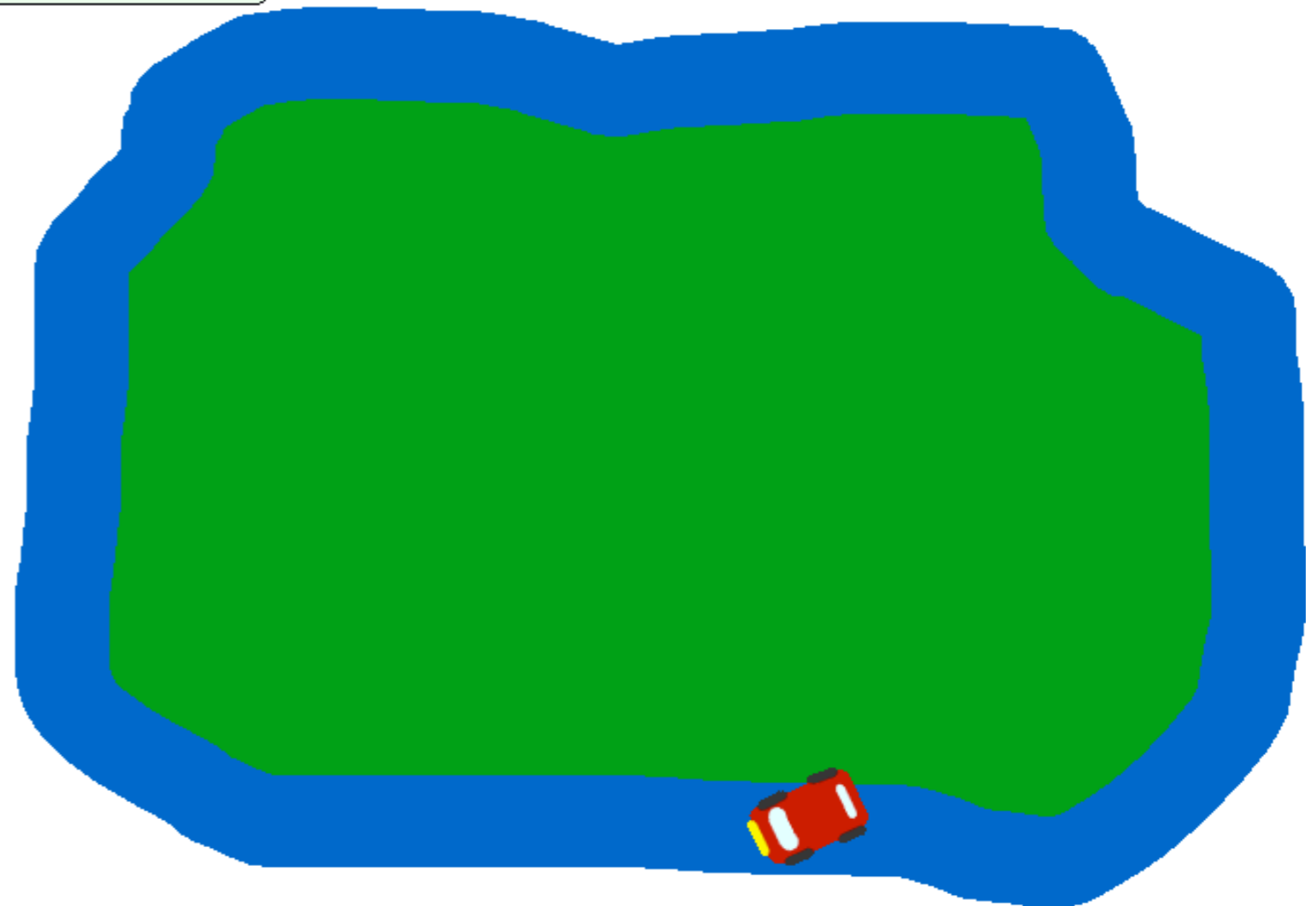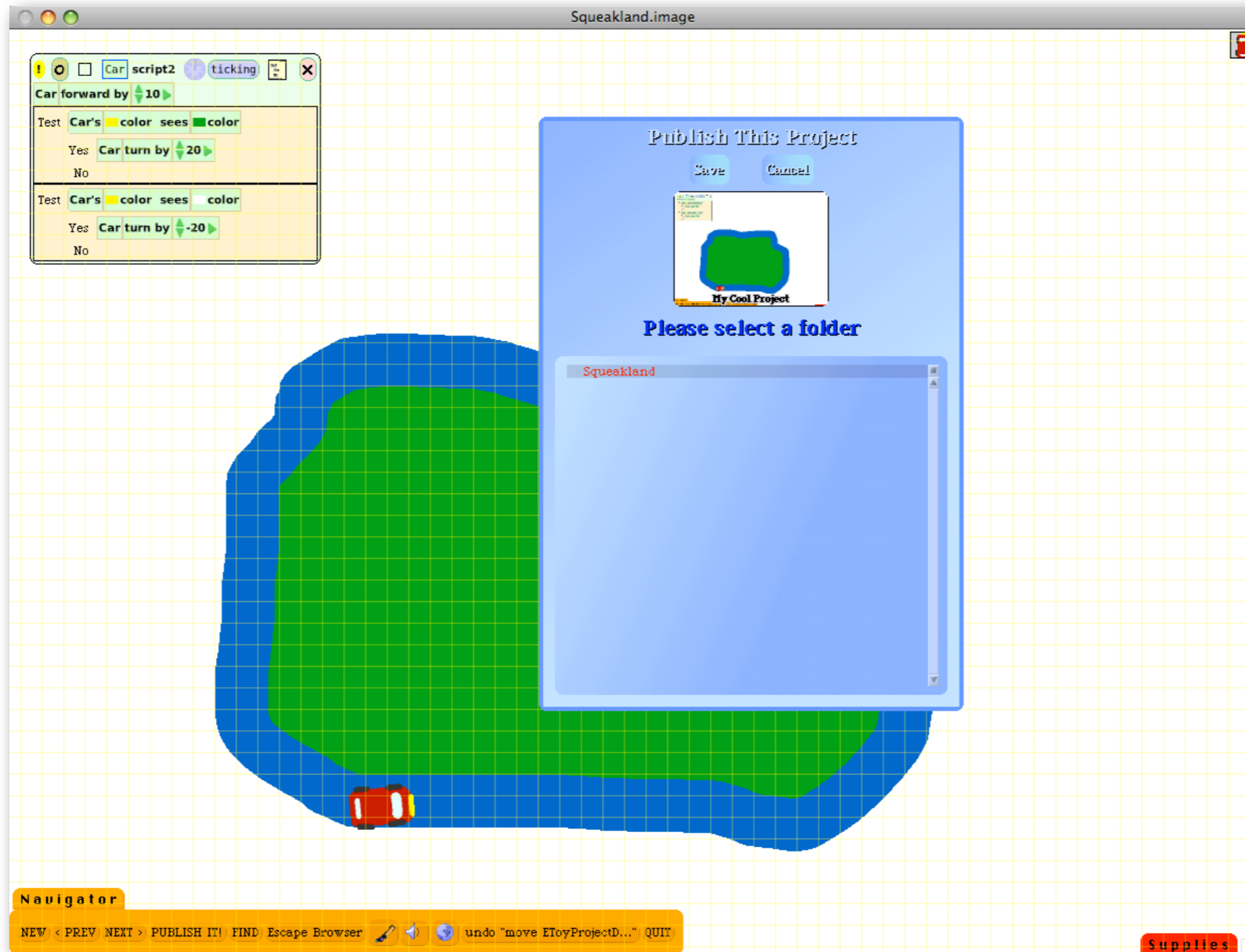
Step 5: script the behaviour
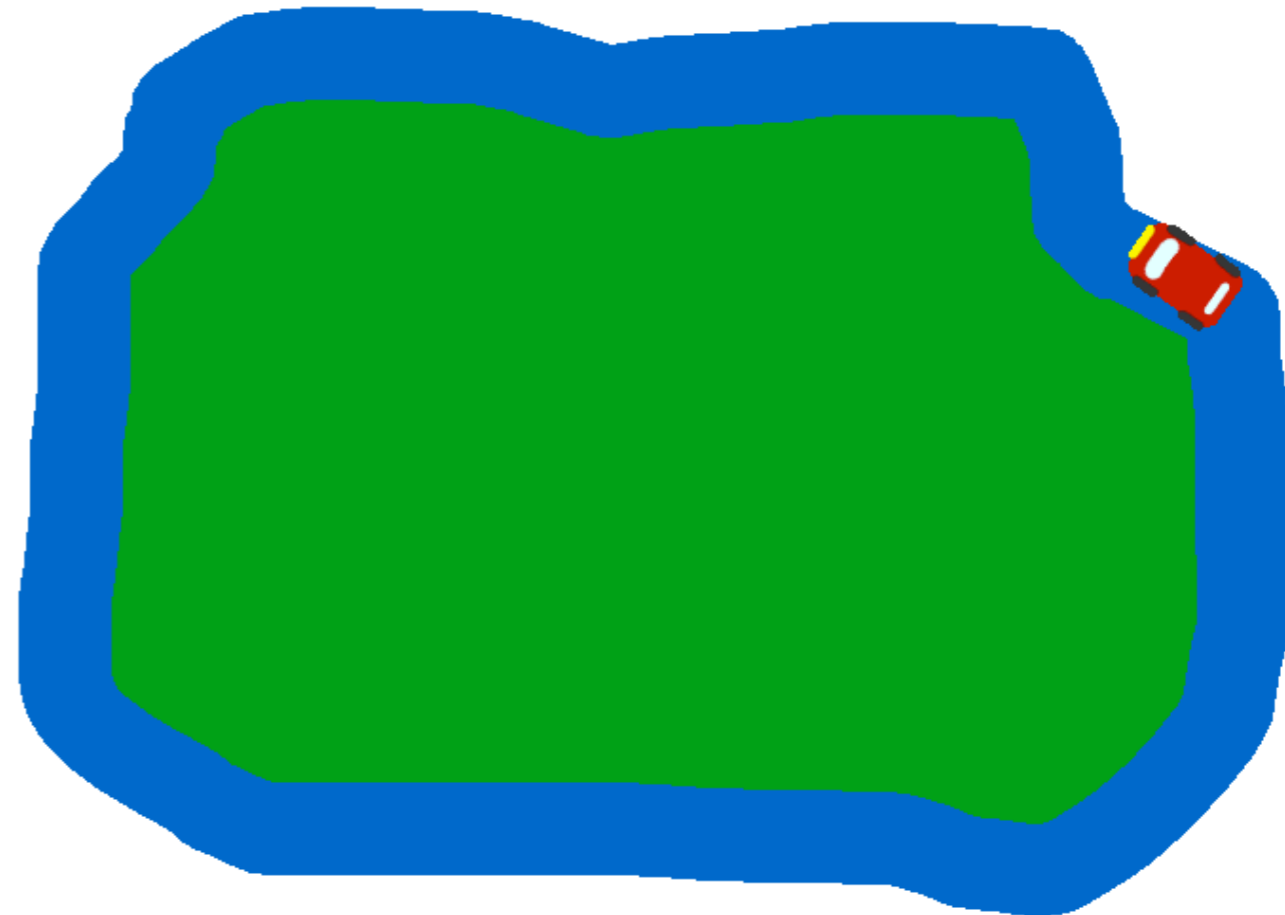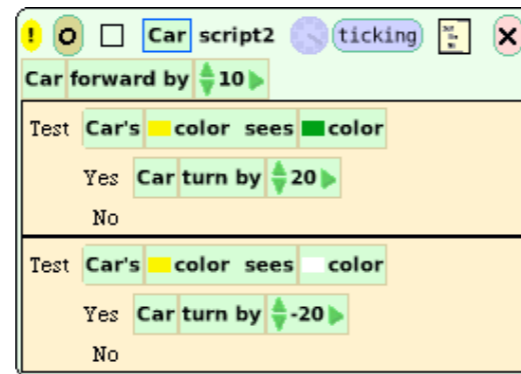
The car always moves forward. If it sees green, it should turn right. If it sees white, it should turn left. In this way it should always stay on the track.

(In practice you will have to fiddle with the parameters to keep it from going too fast, running off the track, and not being able to recover.)

Step 6: publish!

Demo: State Machines as EToys

1. Open the *Connectors* Object Catalog
2. Clone a "Pin"
   - Rename it to State
   - Define "turnOn" and "turnOff" scripts to set the colour
   - Add an "enabled" variable and "enable" and "disable" scripts
3. Connect two States
4. Define the "click" script
   - Adapt "turnOn" and "turnOff" to enable/disable successors
5. Make buttons for On and Off states and curvy connectors
6. Put the buttons in a Button Bar
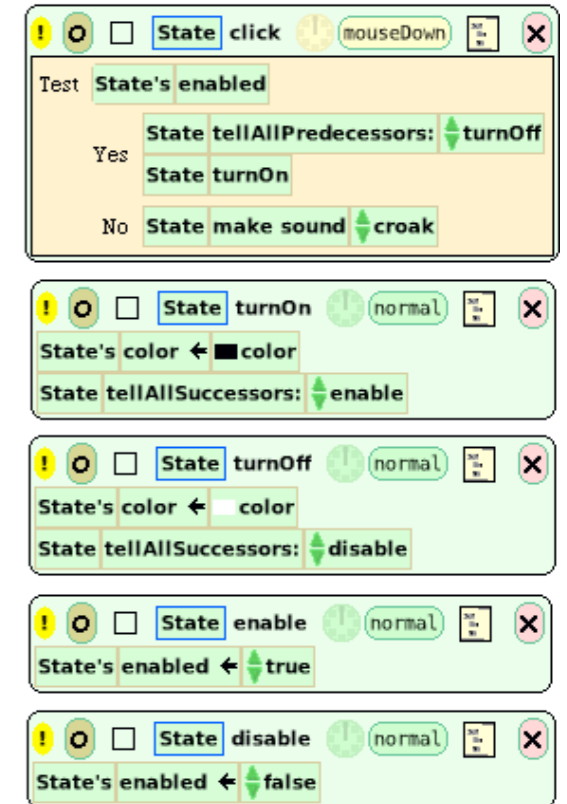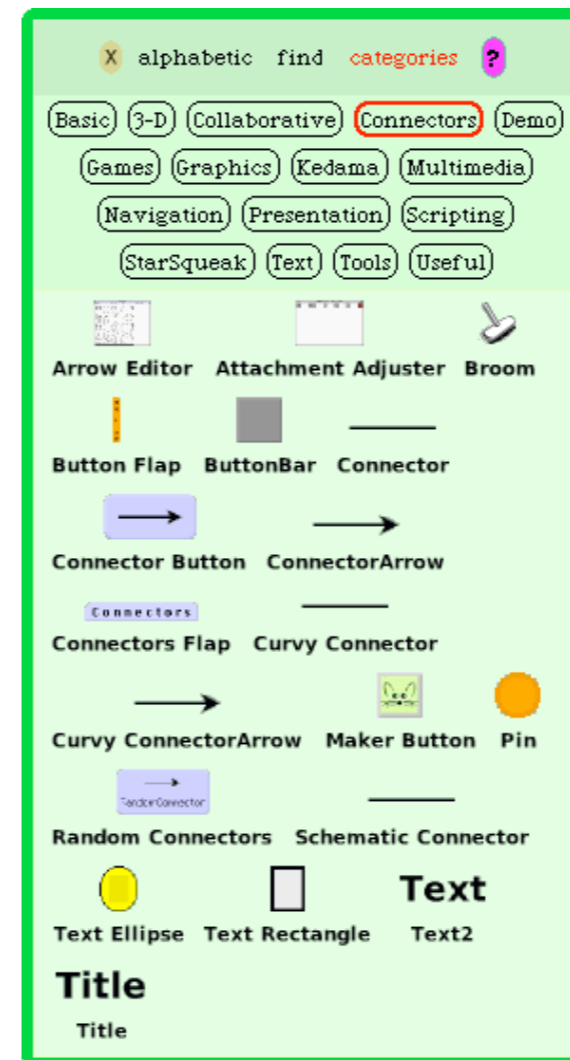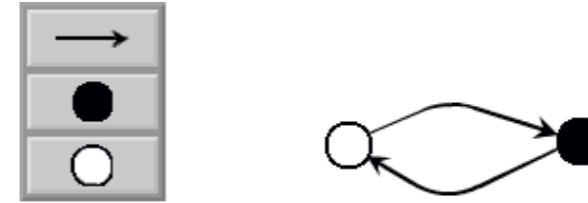
In this demo we develop a factory for constructing finite state machines as graphs consisting of connected nodes. Each graph should contain one black node representing the current state. You may click on any white node that is connected by a transition (an arrow) following the black node. This triggers a state change and causes that node to turn black. Clicking on an illegal node (i.e. not following the current state) will just generate an error noise.
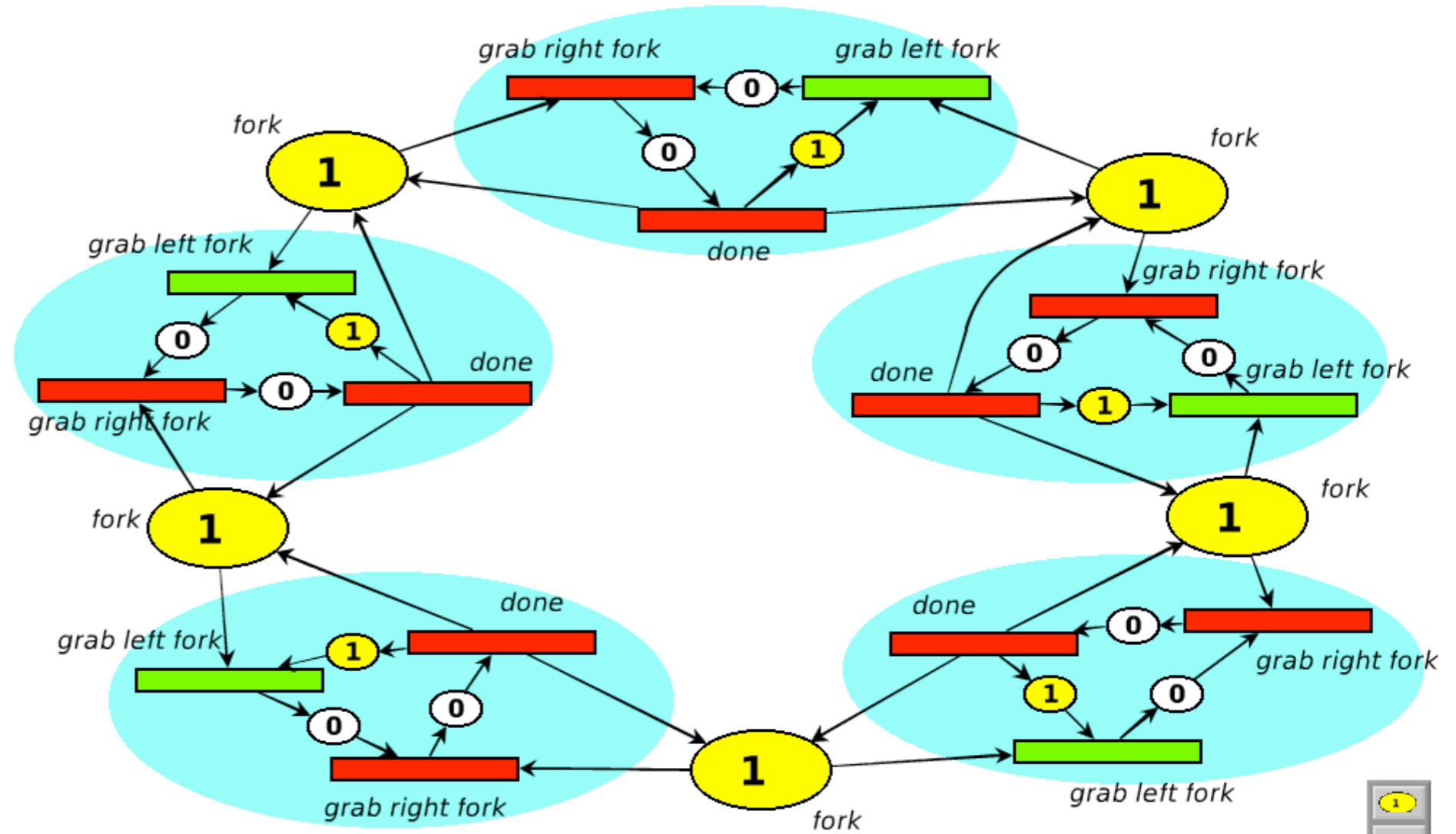
A canned video is also available for this demo.

http://scg.unibe.ch/download/Demos/Videos/eToys-stateMachine-demo.mp4

EToys idioms — Visible Factory

Visible Factory:
Store the prototype of an object in some place which makes sense to the end user.

Dining Philosophers

This demo is a Petri net simulator used in a MSc course on Concurrent Programming.

http://scg.unibe.ch/teaching/cp

http://scg.unibe.ch/download/petitpetri/

Like the state machine demo, it uses the idiom of a "visible factory" to generate the elements of a Petri net. This idiom and others are documented in the paper, "*Idioms for Composing Games with EToys*".
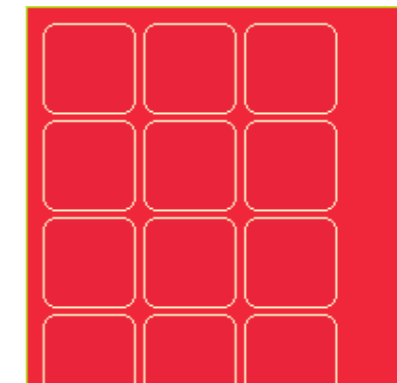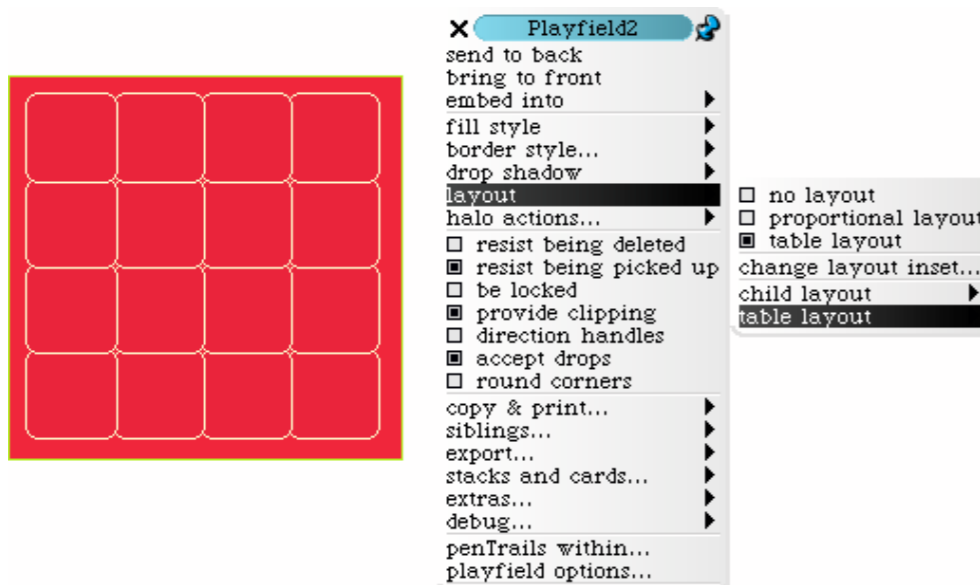
http://scg.unibe.ch/archive/papers/Gael06aC5.pdf

EToys idioms — Connected Neighbours

Connected Neighbours: encode neighbourhood relationships between objects by letting them overlap.
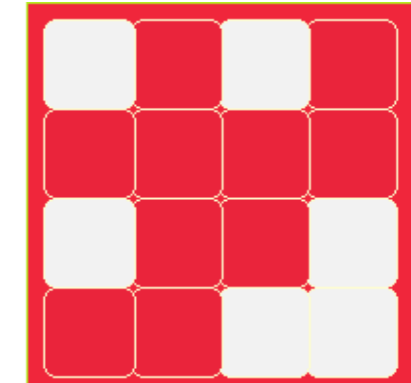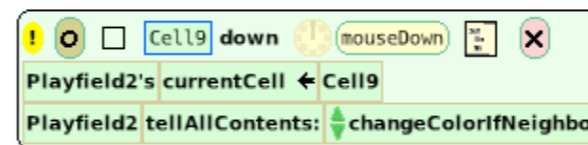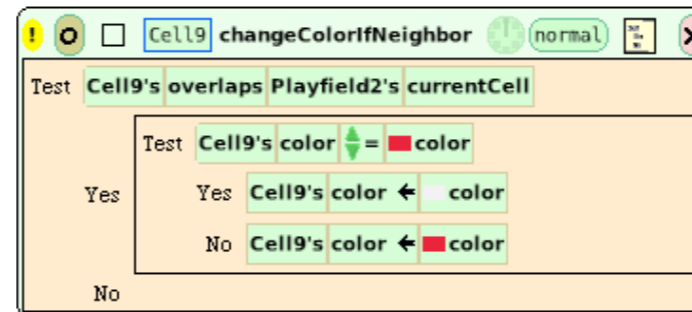
Example: Lights Out

The table layout of the "Lights Out" Playfield is adjusted so that diagonally adjacent Cells do not overlap!

EToys idioms — Visual Cursor

Visual Cursor: Store a selected element of a playfield in a "cursor".

Playfield and Holder objects have a "<u>cursor</u>" which keeps track of (the index of) the currently selected object that they contain.


Example: The 15 Puzzle's cursor keeps track of the current empty Cell. Each Cell keeps track of its current number. If a move is legal, the number is transferred to the empty cell, and the clicked cell becomes the puzzle's currently empty cell.

EToys idioms — Intelligent Environment

Intelligent Environment:
Encode otherwise implicit behavior of an object into another object of the environment.

Example: The winning condition for TicTacToe is computed with the help of StoneCounter objects for each row, column and diagonal.
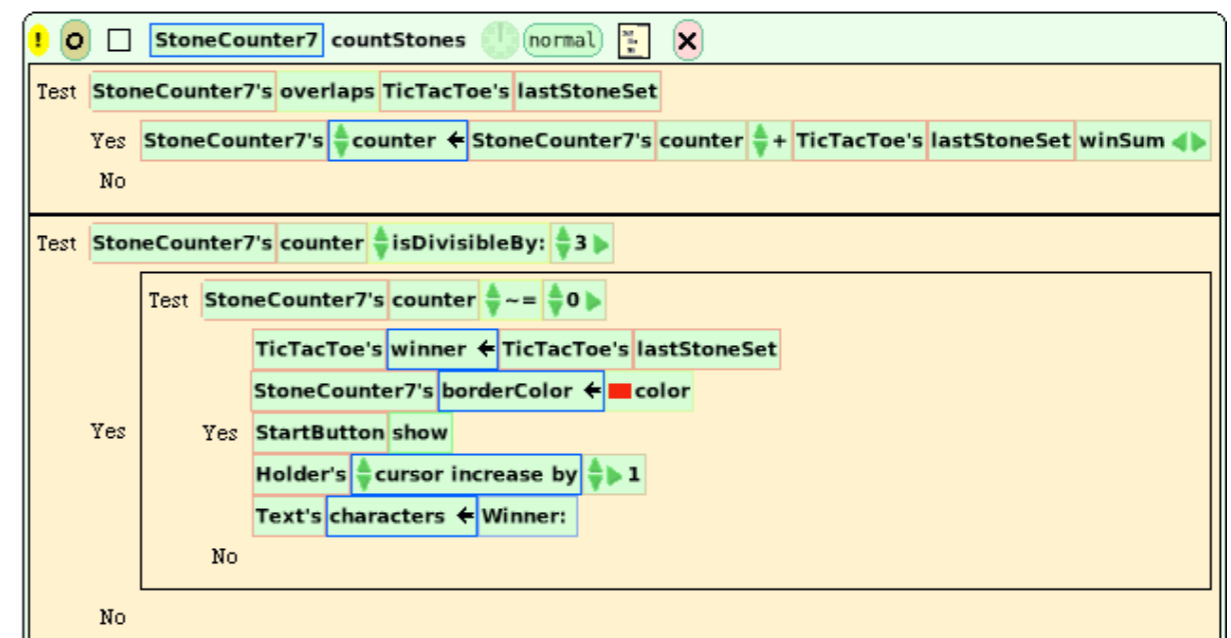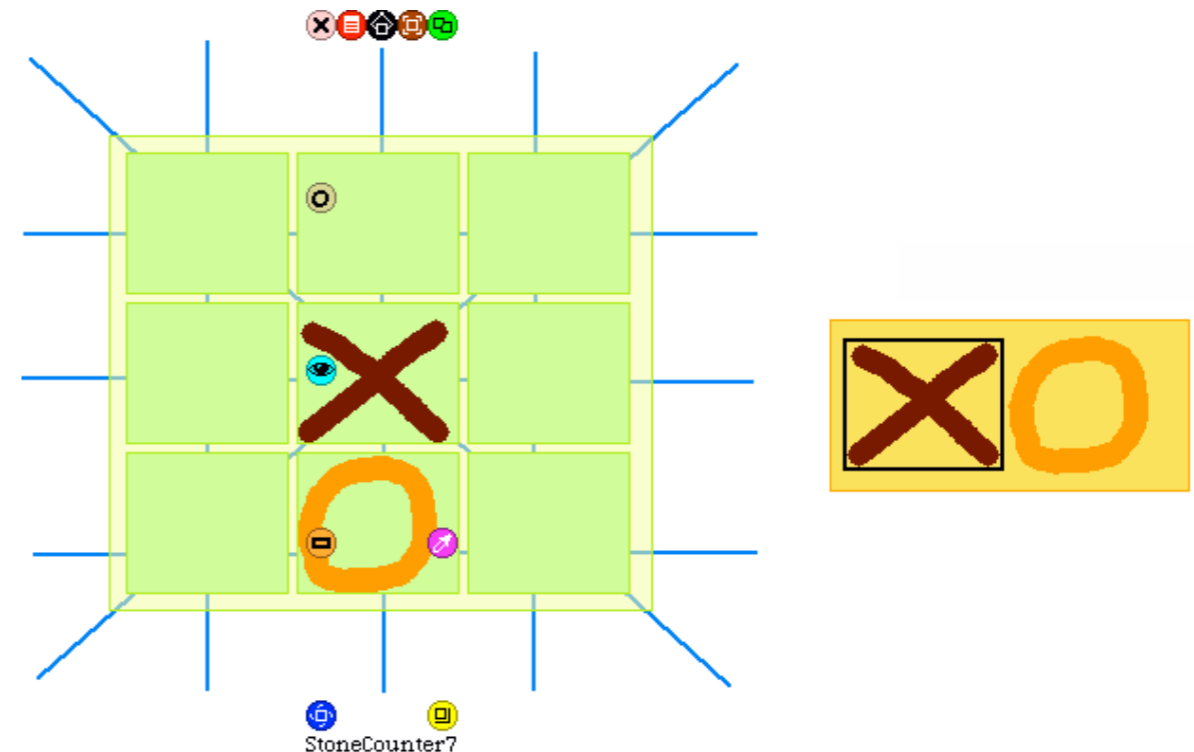
EToys critique

The Good

> Unified GUI and model

> Tile composition

> Rich toolkit (holders, connectors …)

> Prototype-based (encourages exploration)

> Variables typed by example

> Generates Smalltalk behind the scenes

> Highly expressive (e.g., not limited to animation)

The Bad

> Hard to extend with Smalltalk

> Limited event set

> No matrices

> Projects are stuck in Squeak 3.0

> No arithmetic expressions

> No debugger

> Command menu structure is strange (e.g., "misc")

> Scripts can't "return" a value

The Ugly

> Etoys code pervades Squeak

www.emergent.de/pub/smalltalk/squeak/projects/EtoysLecture.pr

45

Scratch

More professional appearance than Etoys, but less expressive. (Communication by broadcasting; no connectors.)
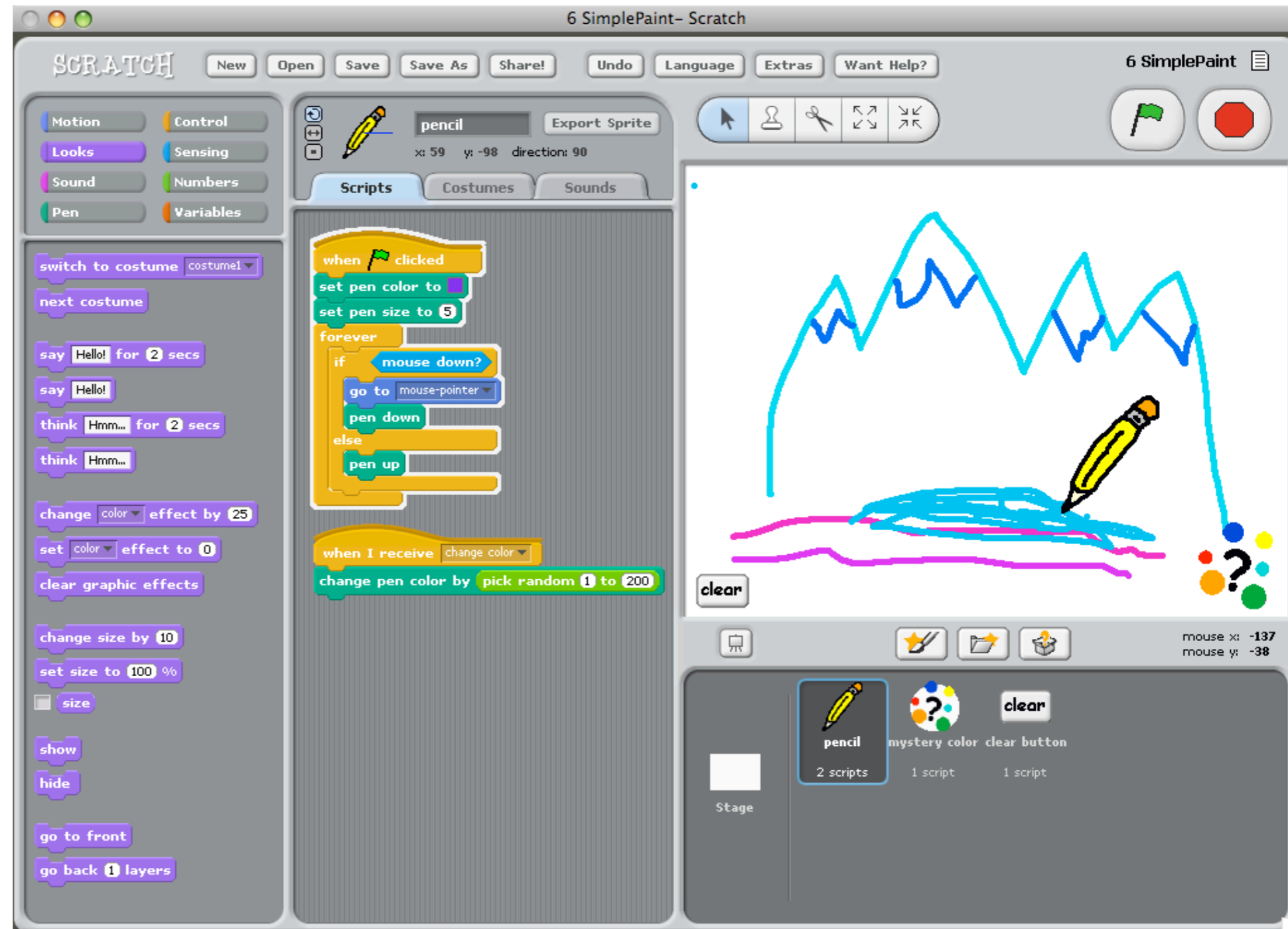
Scratch was developed at the MIT Media Lab and is widely used to *teach children how to program*.

As with eToys, programmers compose scripts from pre-packaged tiles to animate graphical objects. Although the GUI is more appealing than that of eToys, Scratch is strictly less powerful due to its underlying computational model of broadcasting events. It is not possible, for example to implement the Petri net interpreter in Scratch.

What you should know!

- ✎ How would you define Visual Programming Languages?
- ✎ Is Excel a VPL?
- ✎ What are the key paradigms of VPLs?
- ✎ What is the relationship between ADLs and VPLs?
- ✎ What role do types play in VPLs?
- ✎ What are typical ways of classifying VPL?
- ✎ What are the capabilities and limits of EToys?

Can you answer these questions?

✎ *Why has Smalltalk been used to implement many VPLs?*

✎ *Do Naked Objects violate the principles of MVC?*

✎ *What are practical applications of graph transformations?*

✎ *Why do so many VPLs seem to be based on dataflow?*

✎ *Why are all mainstream programming languages textual?*

creative commons

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

   Share — copy and redistribute the material in any medium or format
   Adapt — remix, transform, and build upon the material for any purpose, even commercially.

   The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

   Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

   ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

http://creativecommons.org/licenses/by-sa/4.0/