Gadget

Survey

1

Stage 1 of Gadget examined a large swathe of prior works related to the Gadget project. We conducted an open-minded and thorough survey of related work, from Smalltalk to Minecraft. We surveyed 48 different systems, producing one page visual distillations for each.

Chaim Gingold • Wed Jul 05 2017

Gadget

Purpose. The Gadget project combines cutting edge ideas from programming languages and game design to invent new tools for novices learning to code as well as expert users. It is predicated on the observation that some of the most powerful ideas in the history of computers—from interface design to programming languages—have come from making systems more tangible, alive, playful, and accessible to children. Drawing on influences from Smalltalk to Minecraft, Gadget seeks to build captivating play experiences that transform users into proficient and creative computational thinkers. But Gadget is more than a playful tutorial; it aims to transform the experience of programming itself.

1
2
3
4

Approach. The project is divided into four stages: survey, articulating design values, prototyping, and design. In the first stage, we conducted an open-minded and thorough survey of related work, from Smalltalk to Minecraft. We surveyed 48 different systems, producing one page visual distillations for each. In stage two, five high level design values were distilled from the review: a quality of world-ness, linked representation, tactile, personally meaningful, and directed and undirected activity. In the third stage of the project, we will put these design values into action by building and testing prototypes that push the envelope in programming environment design. In the fourth stage we will summarize our learnings in the form of a design for a new computational world.

Who. The Gadget project builds upon the combined background and expertise of Dan Ingalls and Chaim Gingold. Among his many seminal contributions to computing, Dan Ingalls has contributed to making programming more tangible, alive, and open to creative improvisation (e.g. Squeak and Lively). Chaim Gingold brings to the project expertise in designing simulations, play experiences, and creative tools. Recent projects include using simulation toys as book illustrations (Earth Primer), and investigations into diagrammatic representations of software (Ph.D. dissertation).

Chaim Gingold
July 05, 2017
YCR / HARC / Lively

“I fell in love with the gears.”

Survey

“You can be the gear, you can understand how it turns by projecting yourself into its place and turning with it. It is this double relationship—both abstract and sensory—that gives the gear the power to carry powerful mathematics into the mind. In a terminology I shall develop in later chapters, the gear acts here as a transitional object.”

“I fell in love with the gears.”

“My thesis could be summarized as: What the gears cannot do the computer might. The computer is the Proteus of machines. Its essence is its universality, its power to simulate. Because it can take on a thousand forms and can serve a thousand functions, it can appeal to a thousand tastes. This book is the result of my own attempts over the past decade to turn computers into instruments flexible enough so that many children can each create for themselves something like what the gears were for me.”

—Seymour Papert, *Mindstorms* (1980)

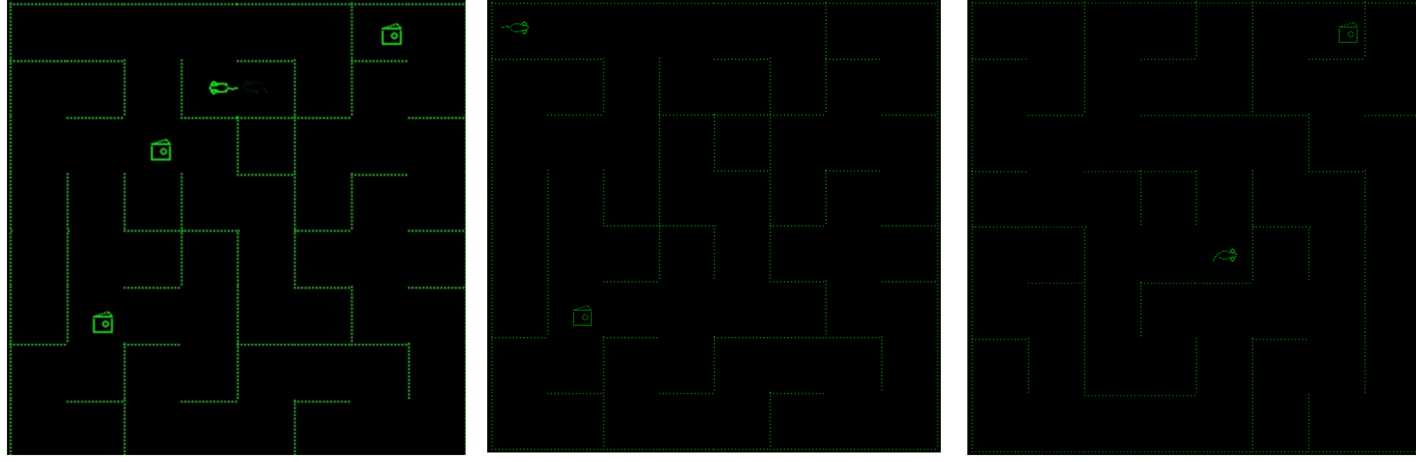
"I fell in love with the gears."

Seymour Papert, *Mindstorms* (1980)

Mouse in the Maze

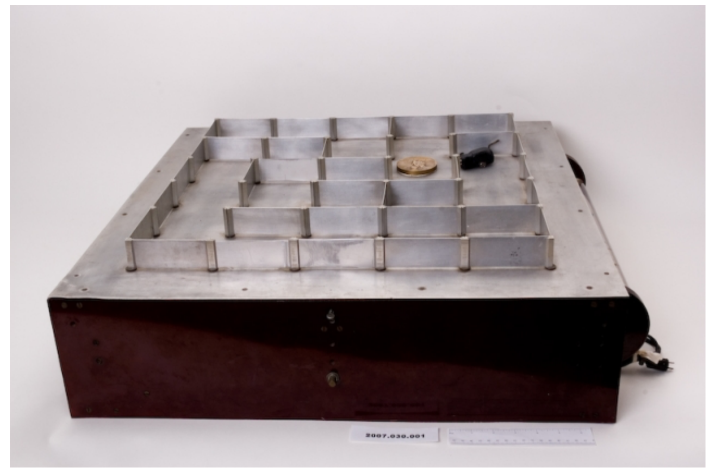
John E. Ward and D.T. Ross (1959), TX-0

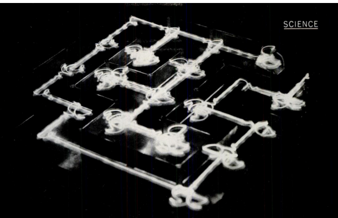
Mouse in the Maze is one of the oldest software toys. Using a light pen, players would design mazes by erasing and making walls, place cheese, and place a mouse. The mouse would then search for the cheese.

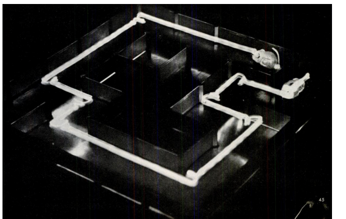


The mouse is perhaps the earliest computer virtual computer characters (predating Weizenbaum's Eliza), and set the stage for the seminal computer game SpaceWar!

The mouse would could get tuckered out if it found no cheese, and needed to be fed to keep its energy going. If cheese was inaccessible, the mouse—after exploring all available space—would complain, "typing out an appropriate comment on the typewriter," and be discouraged until fed again (Ward 1959). Graetz describes a tipsy mouse in a variant mode with martinis not cheese (Graetz 1981).





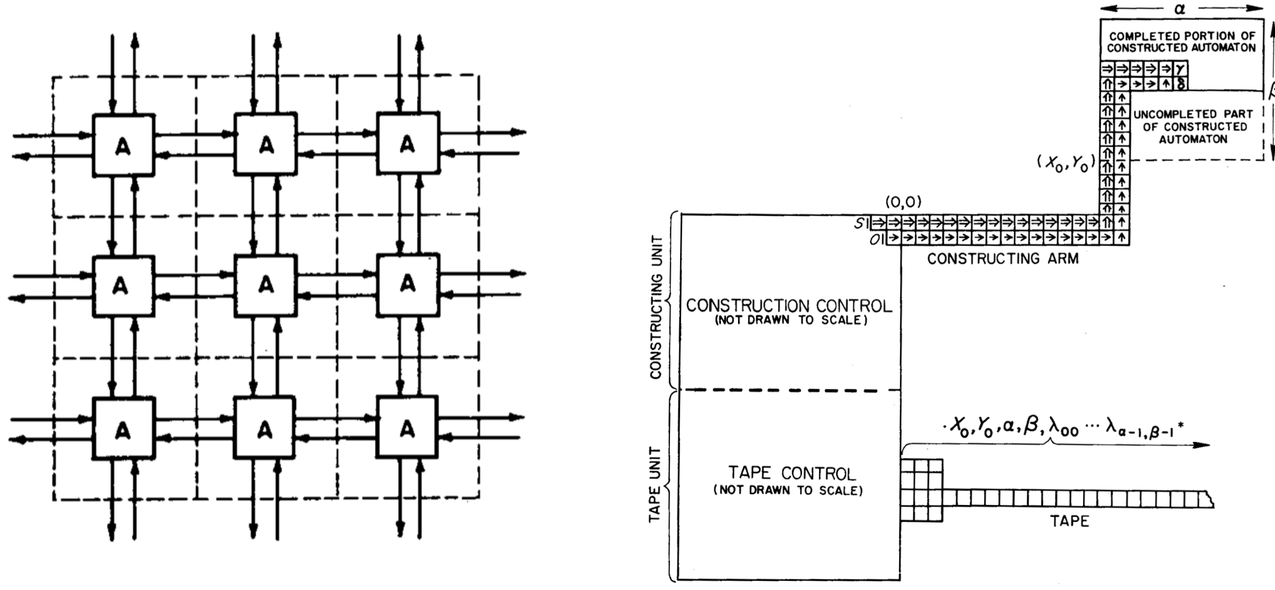


Probably inspired, in part, by Theseus, Shannon's 1952 electromechanical maze solving mouse. Theseus was an early effort to make problem solving machines—Shannon would later create one of the first chess playing computers (Caïssa).

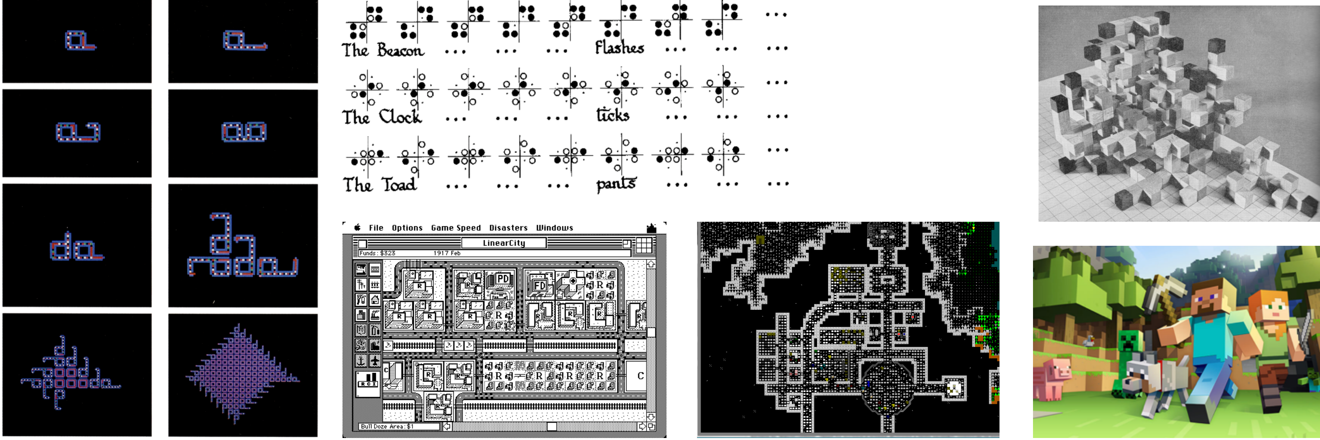
Cellular Automata

(1952–)

Cellular automata are a simulation tradition whose roots lie in a synthesis of **biology and computation**. John von Neumann wished to create a mathematical model of biological self-reproduction, and following a suggestion by Stanisław Ulam, settled upon a lattice of interacting identical elements as a modeling substrate—not unlike the discrete models used for computer weather simulations. (Turing, also, used a similar model for thinking about morphogenesis in 1952).



Space itself is a computational substrate, a mesh of identical interacting components. Higher level structures, for the example the self-reproducing machine above, are said to **embedded** in the cellular space. One of the most famous cellular automata is John Conway's Game of Life, which beautiful exemplifies the **emergent** potency of such systems.



The highly **observable** nature of such systems, coupled with their **representational plasticity**, and emergent potential, affords a high degree of player **agency** and a flexible variety of applications, from biological reproduction and physics to game worlds.

Sketchpad

Ivan Sutherland (1963), TX-2

Sketchpad is a seminal program in the history of computing, combining and introducing a variety of ideas: graphical man-machine communication (**direct manipulation**), instances and masters (**inheritance**), and **constraints**. Sutherland writes that "Sketchpad is *itself* a model of the design process," as designers primarily work with and produce drawings, and the primary concern of design is fashioning within constraints.

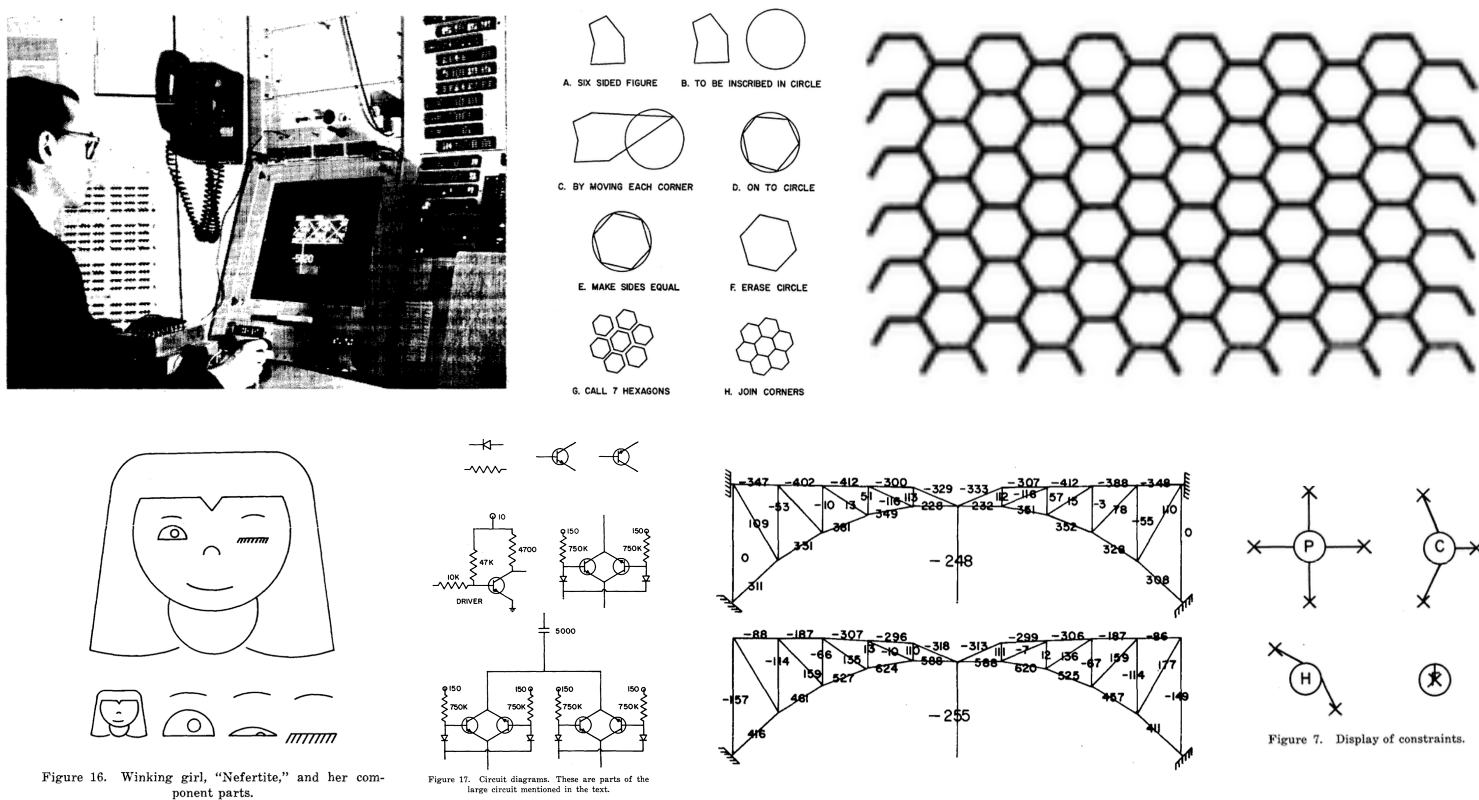
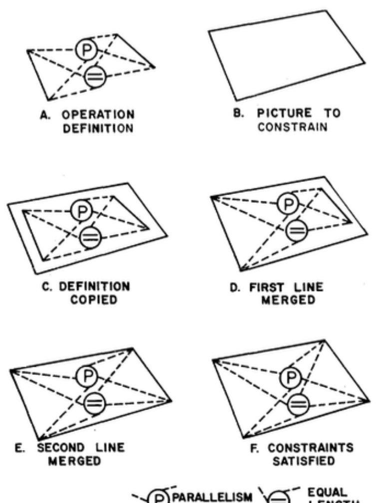


Figure 16. Winking girl, "Nefertite," and her component parts.

Figure 17. Circuit diagrams. These are parts of the large circuit mentioned in the text.

Figure 7. Display of constraints.



Masters can define attachment points, as well as simply be "constraint complexes" applicable to appropriately typed objects, allowing libraries of smart objects to be recursively built up. A switch shows/hides constraints as manipulable screen elements.

Merging—of points, lines, constraints, composite shapes—allows complex objects to be built up. By merging line endpoints polygons can be built up. Merging a "constraint complex" to a shape applies that operation to a shape.

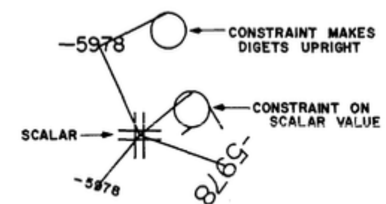


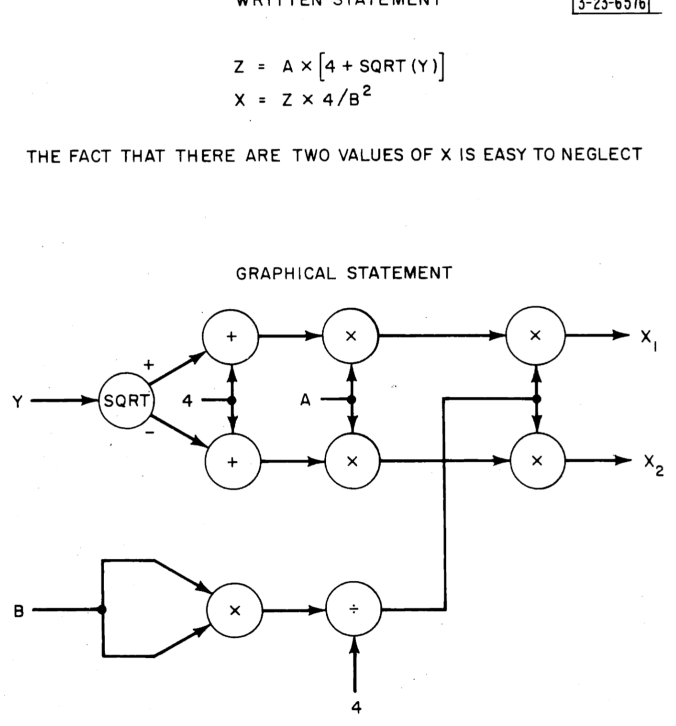
Figure 8. Three sets of digits displaying the same scalar value.

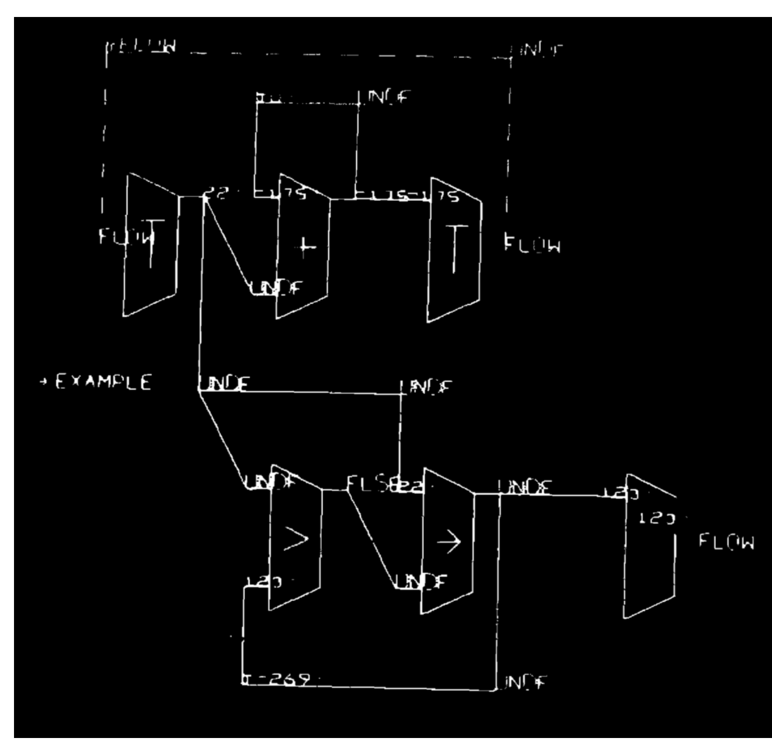
Sutherland, Ivan E. "Sketchpad a Man-Machine Graphical Communication System." Transactions of the Society for Computer Simulation 2, no. 5 (1963): R–3.

On-Line Graphical Specification of Computer Procedures

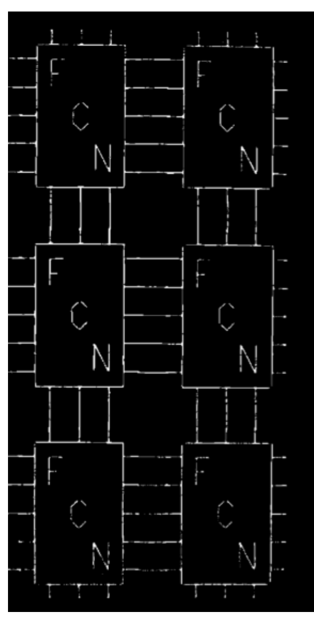
William Sutherland (1966), TX-2

Sketchpad is for graphical representation of data, but OLGSCP is for graphical representation of programs. Programs are represented as data-flow diagrams.

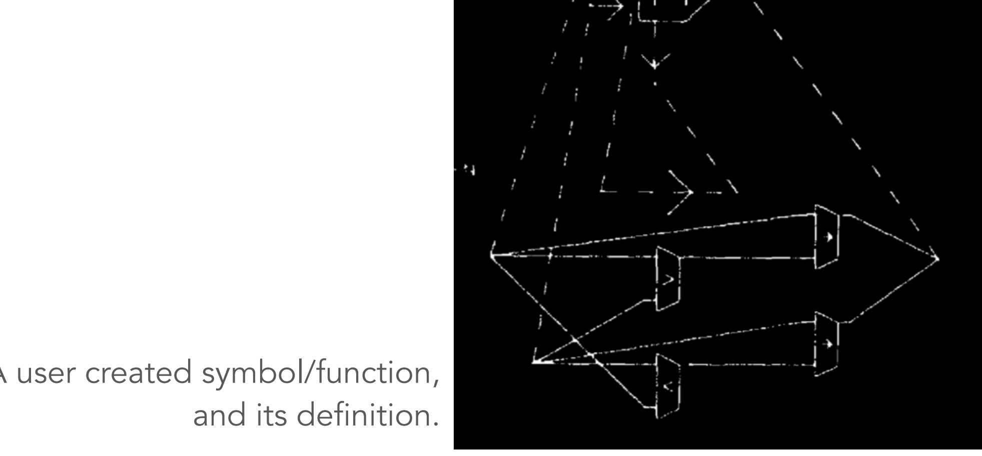




Many ingenious debugging features are offered. Variable values can be shown on the wires. Probes and breakpoints can be inserted into the program.



Automatic connections, inferred by proximity and data type, aid the creation of complex programs.



A user created symbol/function, and its definition.

Logo

Seymour Papert, Wally Feurzeig, Cynthia Solomon; BBN (1967–)

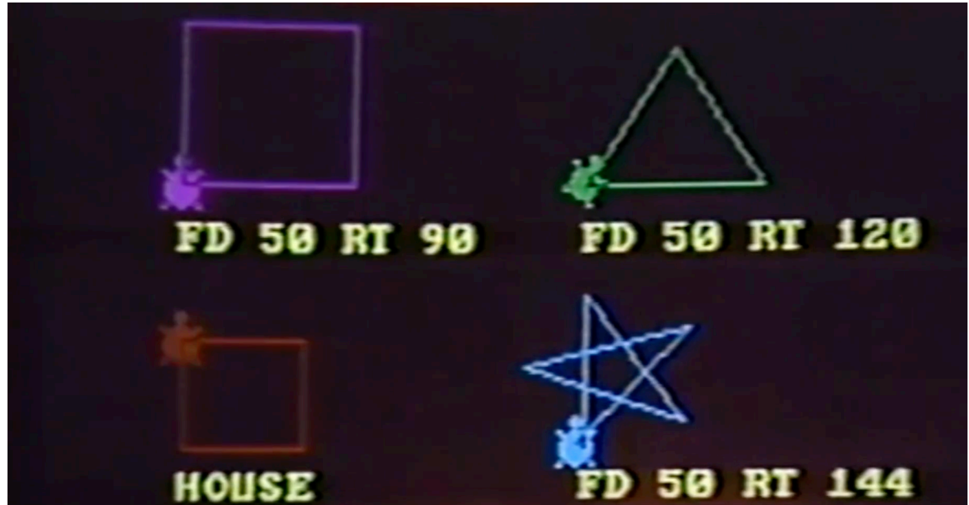
Logo was intended to create a living world, a culture, in which a domain—in this case mathematics—could be easily absorbed by young learners. Just as French is most easily learned by children in France, Papert sought to create a microworld—mathland—in which children could easily learn math.

Early versions of Logo used tangible robots to perform programs—moving around and drawing pictures. Later, a virtual turtle on a computer screen was used. The turtle has "holding power" (it's fun), and affords "playing turtle" (identification). (Papert 1987)

The turtle was inspired by William Walter's autonomous tortoise robots.

Papert argued that linking the abstract and the sensory—for example performing a program or shape—was a powerful way to link multiple representations. He called this body linkage "body syntonic." The turtle functions as a "transitional object" between the self and a domain (Papert 1980).

Image from http://battle-bot.blogspot.co.uk



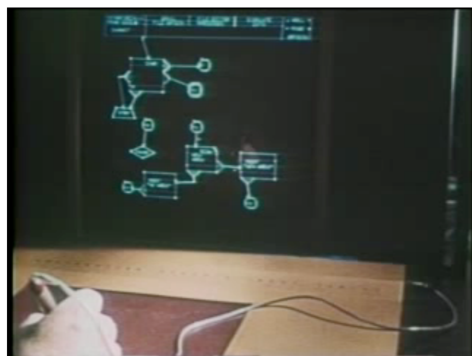
https://www.youtube.com/watch?v=fTO-Ruby-Uo Uploaded by Cynthia Solomon

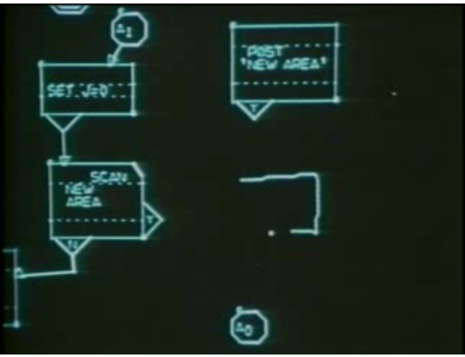
The turtle functions as a vital "body syntonic" link between a person and the domain. A classic example is closing your eyes and walking in a circle to gain an understanding of how to make a circle.

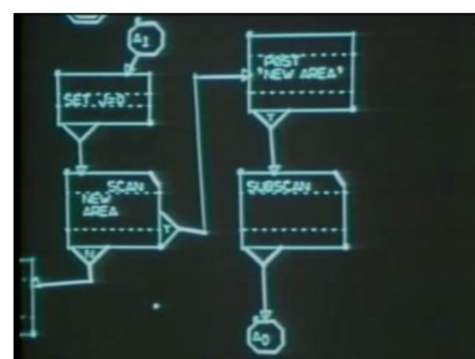
GRAIL (Graphical Input Language)

Ellis, Heafner, Sibley, Groner, and others (1969).

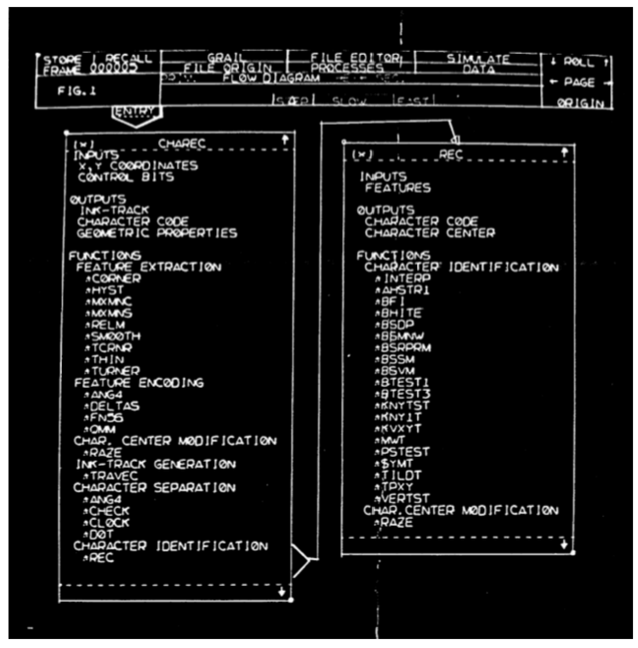
In GRAIL, users draw graphical flow-charts in order to write software. It was used to make sophisticated programs—the GRAIL system, for example, is written in itself. Users draw pictures, write characters, and manipulate virtual objects. It used a tablet and CRT, and was developed at RAND Corporation.

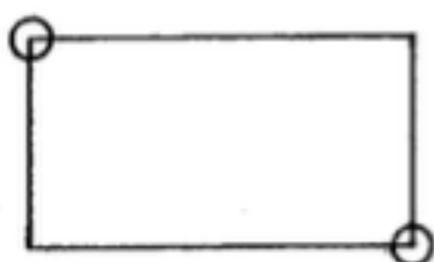






Programs can be compiled and run at full speed, or stepped through with a debugging interpreter that can run the program at variable speeds.





Once they have been drawn and recognized, objects can be moved (top-right handle) and resized (bottom-right handle).

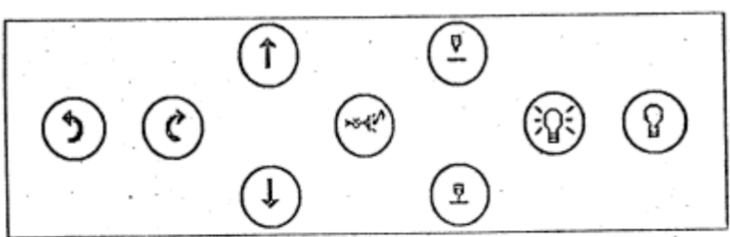
Kay, Alan. "Doing with Images Makes Symbols." presented at the Higher Education Marketing Group, Apple Computer, Inc., 1987. (Video stills.)
Ellis, T. O., John F. Heafner, and W. L. Sibley. "The GRAIL Project: An Experiment in Man-Machine Communications." Rand, 1969.
Ellis, Thomas O., John F. Heafner, and W. L. Sibley. "The GRAIL Language and Operations." Rand, 1969.
Groner, Gabriel F. "Real-Time Recognition of Handprinted Text." RAND, 1966.

TORTIS, The Button Box

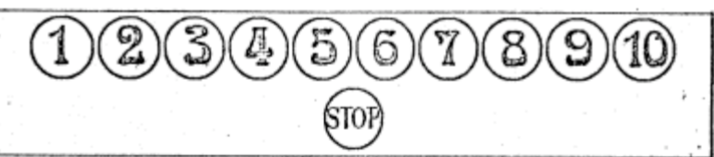
Radia Perlman (1974)

TORTIS stands for Toddler's Own Recursive Turtle Interpreter System. System for introducing programming to pre-literate children for Logo. Boxes with buttons on them are introduced, graduating from puppeteering to programming. As proficiency increases, new boxes are introduced that plug into and expand possible actions.

1. Action Box
- Forward, Back
- Rotate
- Toot Horn
- Pen Down, Up
- Light On, Off

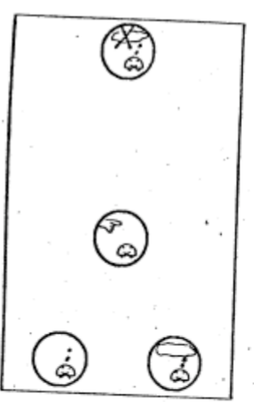


2. Number Box
Push a number before an action, and the action is done that many times. Stop interrupts action.

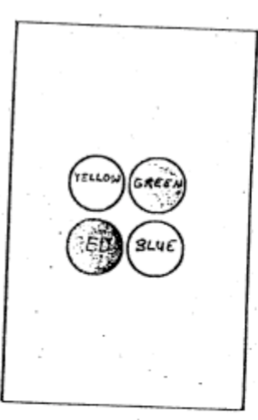


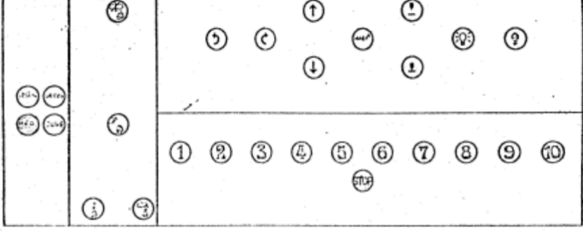
3. Memory Box
- Start remembering
- Stop remembering
- Do it
- Forget it

(Works in conjunction with a display.)

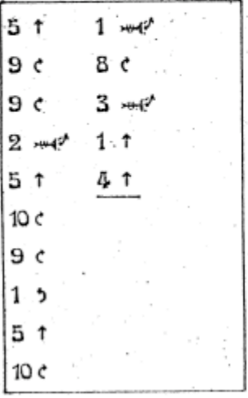


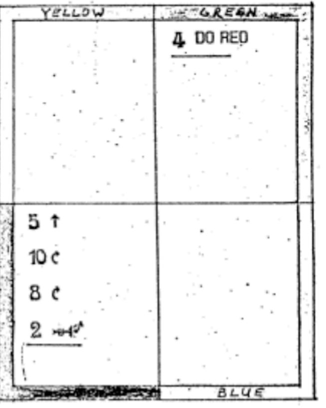
4. Four Procedure Box.
- Remember and playback multiple procedures; allows subprocedures.





All boxes together.





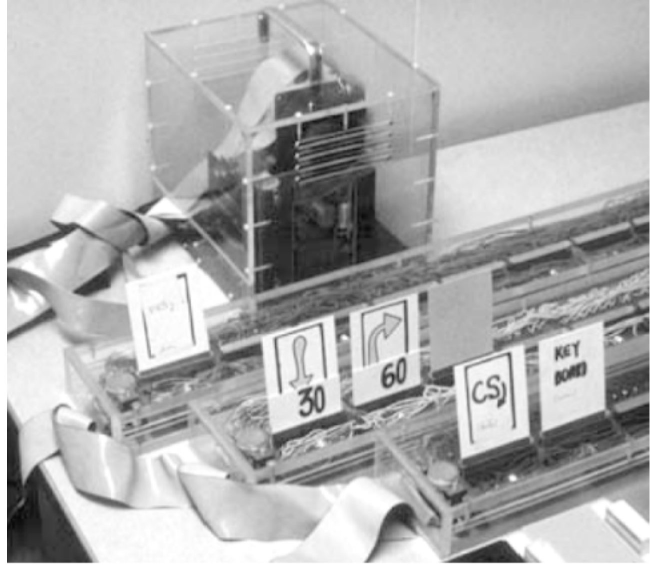
Memory displays for one and four memory boxes.

Images from Perlman (1976), which has some fantastic observations about what worked and didn't work. Photos from http://cyberneticzoo.com/tag/radia-perlman/

TORTIS, The Slot Machine

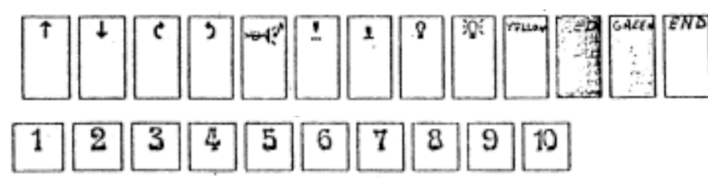
Radia Perlman (1976)

The slow machine design was intended to address some of the shortcomings in TORTIS—that the commands are stored to memory "by the child, not by some magic that occurs when it somehow enters a different mode." (Perlman 1976). She had noted that children attended to the reactive display in Button Box when storing things to memory, and used the memory mode as a way to achieve visual effects on the display—not it's intended use at all. The Slot Machine made the memory elements tangible. Cards allow the language to gradually grow in complexity—like the Button Box.

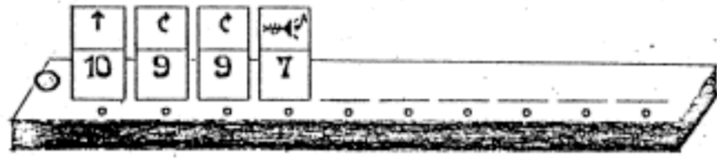


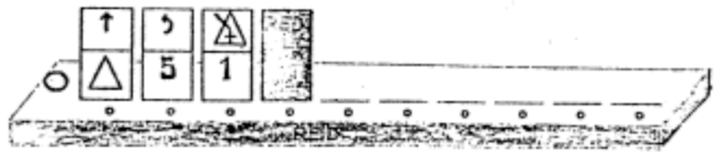
Since the display wasn't needed to sow program memory, Perlman used the screen as the primary output.

Some of the drawings shown here, (from Perlman 1976), are speculative designs.



Action and movement cards.





Program to "draw a square and toot," and another to draw a spiral.

See also Morgado et al. (2006) Radia Perlman – A pioneer of young children computer programming.
Photogram from Morgado et al. (2006), other images from Perlman (1976).

SimKit

Adele Goldberg and the Smalltalk team (1976)

Smalltalk job shop simulation tool made for end-users (executives). Each executive participant had a private machine and tutor.

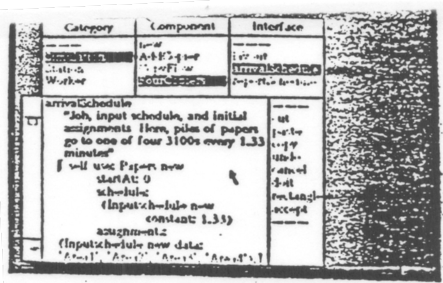
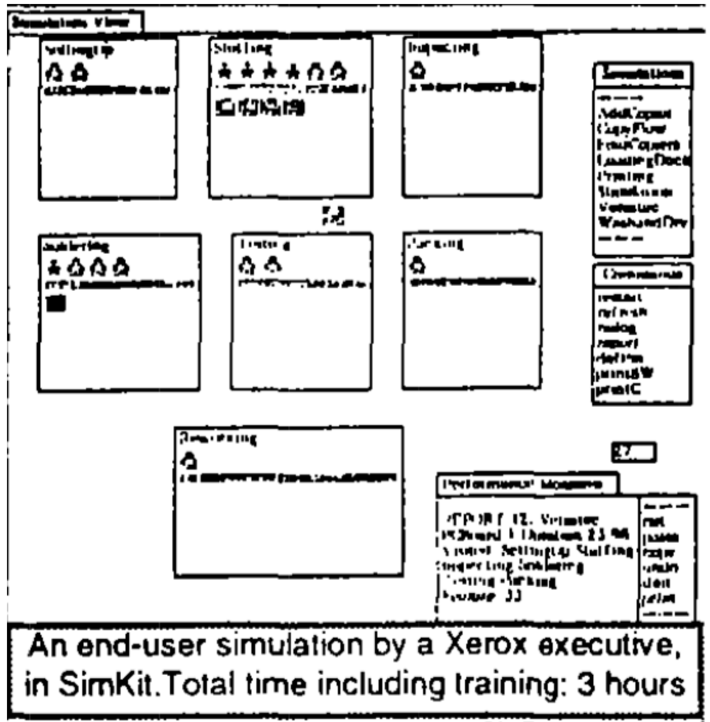


Figure 1. A filtered browser for manipulating simulations



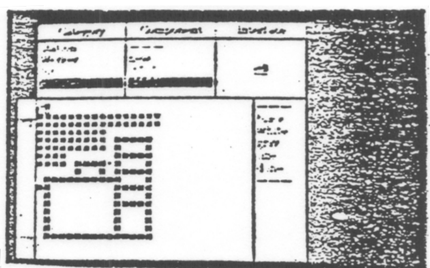


Figure 2. The browser includes icon editing

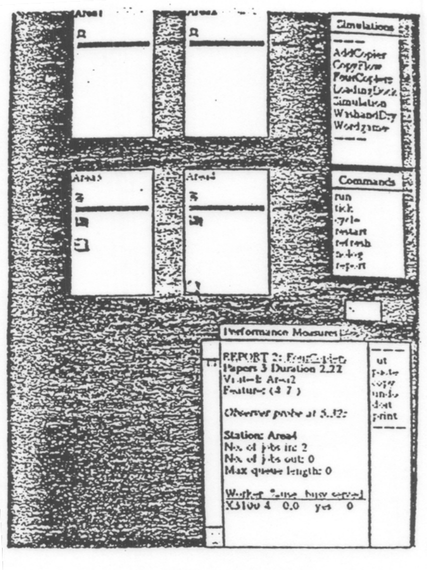


Figure 3. Example of the simulation kit user interface: menus, report window, stations, workers and jobs

Notable features:

- Animating graphics.
- Smalltalk errors translated into meaningful simulation events.
- Mouse tutorial via customization of text size for readability on initiation.
- Custom class browser for four SimKit classes: station, worker, job, report.

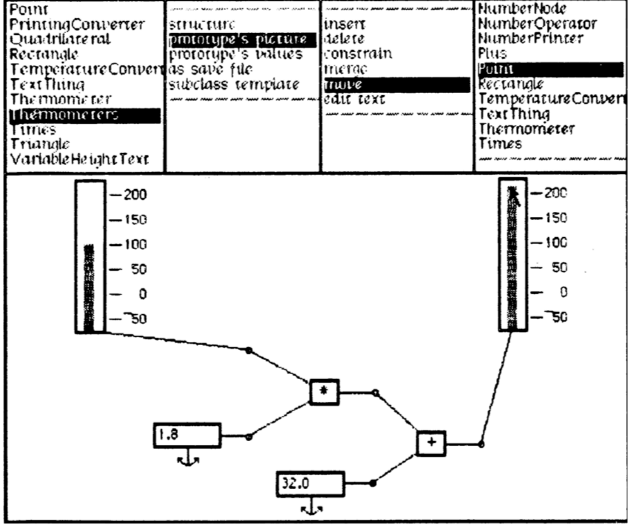
Information on SimKit is meager. Best descriptions are in:

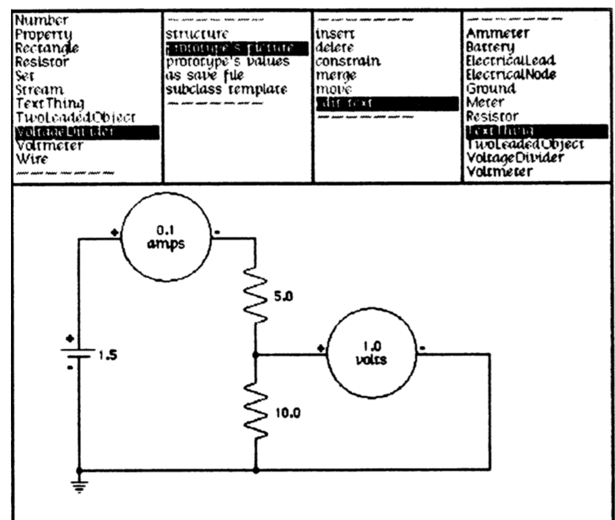
Kay, Alan. “The Early History of Smalltalk.” SIGPLAN Not. 28, no. 3 (March 1993): 69–95. doi: 10.1145/155360.155364.
Kay, Alan. “Learning Research Group Report. January - June 1978.”

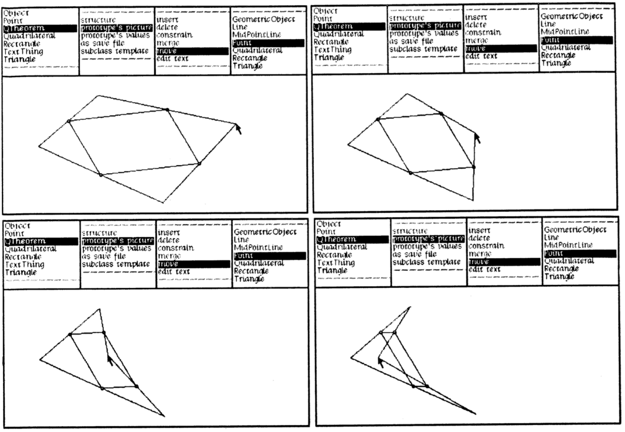
ThingLab: A Constraint-Oriented Simulation Laboratory

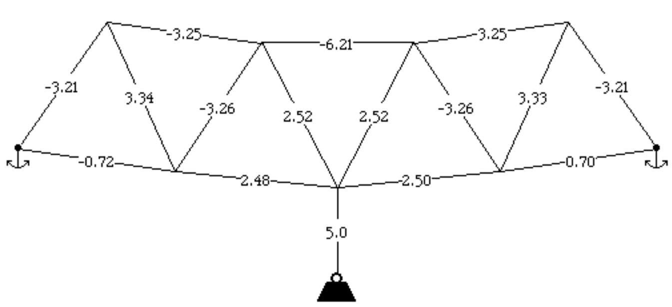
Borning, Alan 1979

Built on Smalltalk









http://esug.org/data/HistoricalDocuments/ThingLab/ThingLab-v.html

Highlights

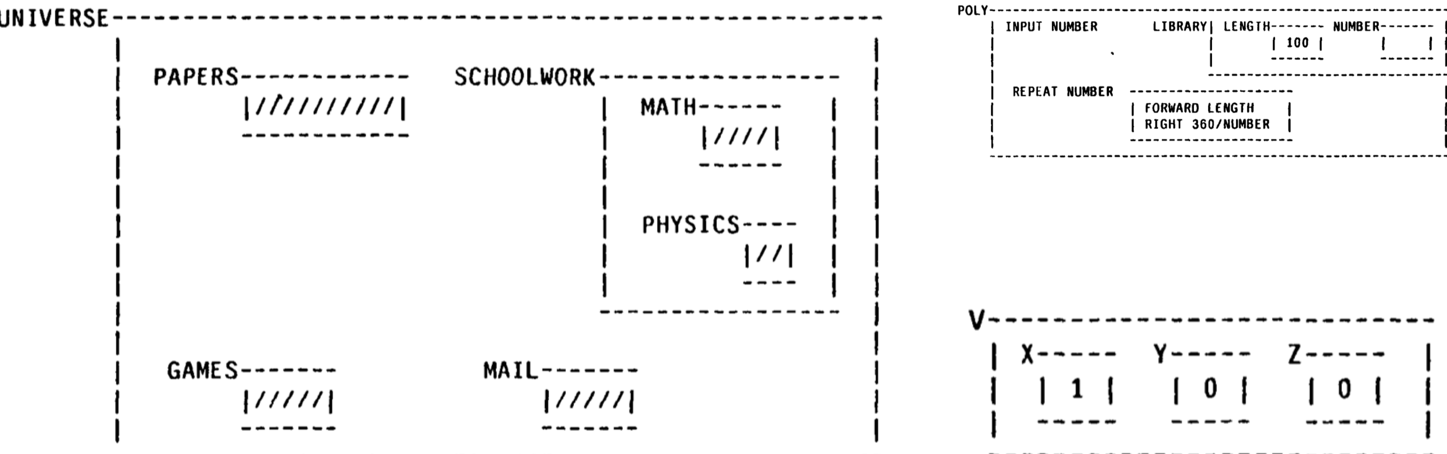
- Constraint based
- Bidirectional data flow (follows from constraints)
- Spatial
- Lends itself to multiple domains; a kit of kits.

Examples include electronics, structural engineering (bridge), geometry (proof-ish), and algebra (Farenheit-Celcius converter).

Boxer

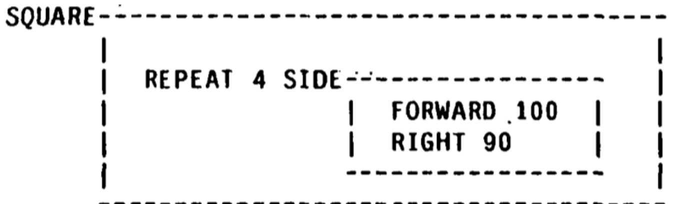
Andrea diSessa (1982)

Boxer is a programming environment that employs a **comprehensive spatial metaphor** for everything: boxes.



In Boxer, "**[o]bjects <u>are</u> their visual representation**," and behave according to "**naive realist**" rules (like Rocky's Boots). Boxer is predicated on the idea of "**surrogates**"—"replacement machines" you think and predict with (for example a physical stack of things as a computational stack.) The interface is built around **editing and browsing,** extending what diSessa thinks is one of the most powerful ideas in Smalltalk, its browser, which allows you to dive into and inspect anything.

Boxer is **environment centric**. It is composed of places with procedures and data—environments that one can experiment in. Boxes can be data, procedures, graphics, and ports. Ports are wormholes into boxes located elsewhere, a powerful idea that enables GUI windows as well as scoping mechanics.



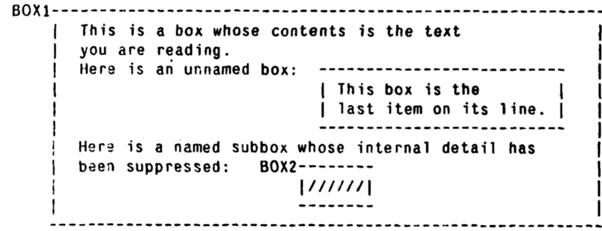


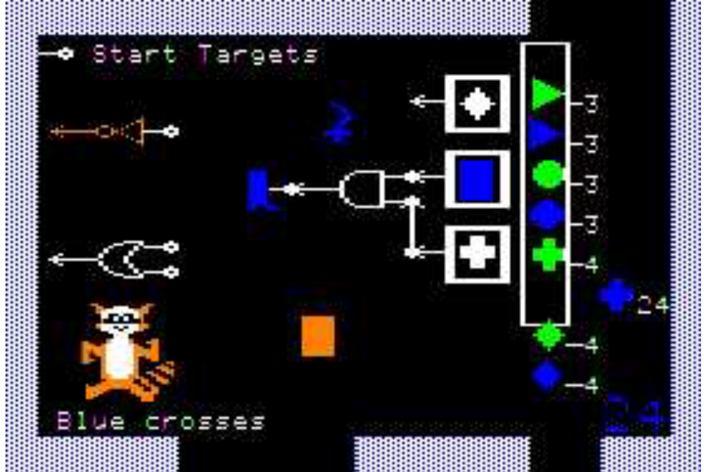
Figure 3-1: A box with contained boxes.

Questions

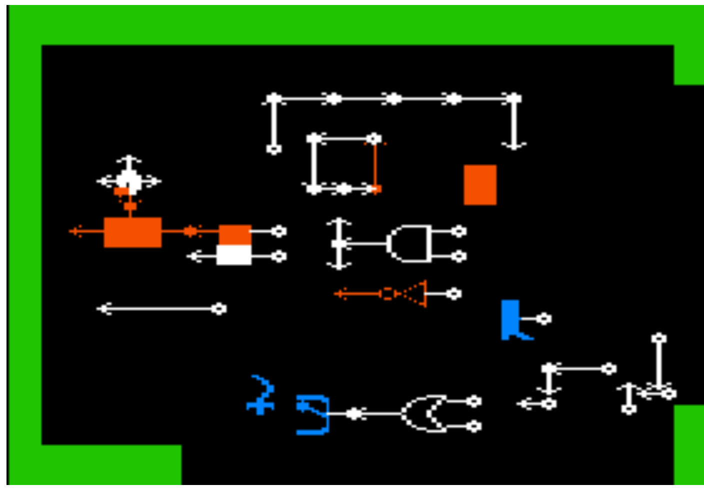
- What sense of world-ness is shared with something like Rocky's Boots?
- Is "naive realist"—which sounds like Rocky's Boots—another way of saying direct manipulation? If not, why not? Is it more general? Specific? Orthogonal?
- Can this blend smoothly into a spreadsheet like representation? (grids of boxes)

Rocky's Boots

Warren Robinett and Leslie Grimm, 1982
Apple II (and other platforms)

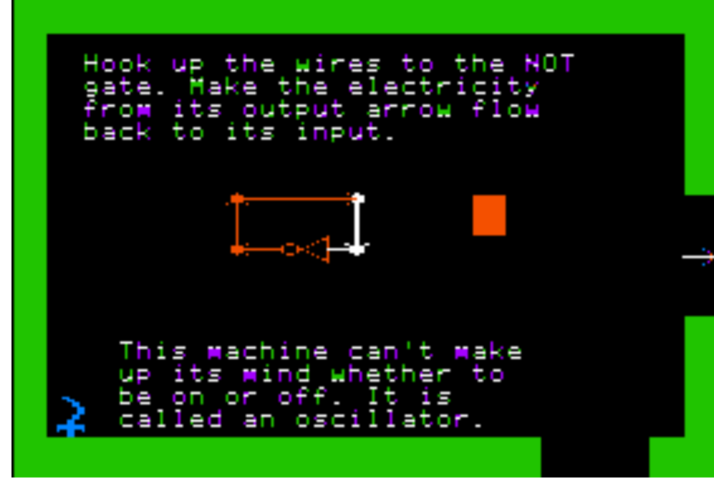


One of the many carnival-like game puzzles. Design a circuit that recognizes blue crosses. When boot is activated it kicks the shapes.



Loose parts create a sandbox effect.

A spatial journey playfully introduces the game.

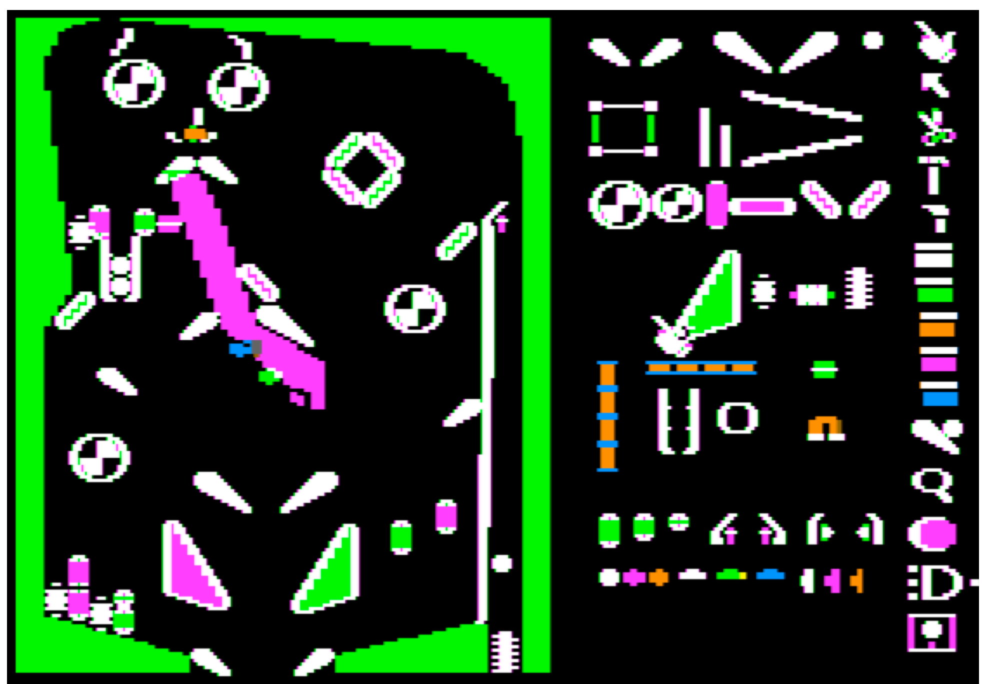


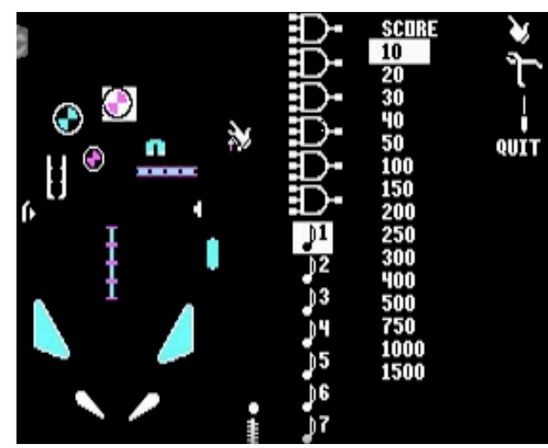
As in HyperCard, even the explanatory materials can be played with—taken apart, recombined, and transformed.

Highlights:
- Logic circuit construction set.
- Multiple ways to engage:
  - As a spatial world to explore (like Robinett's Adventure)
  - As a sandbox—build and experiment
  - As a game. Build circuits that satisfy recognition constraints.
- Gentle on-ramping; traverse spatial world to learn how to play.
- User interface is hampered by a lack of mouse (even more so in emulation, perhaps.)

Pinball Construction Set

Bill Budge, 1982 (BudgeCo, Apple II), 1983 (Electronic Arts, other platforms)





Highlights:

- Design and play simulated pinball machines.
- Established "construction set" and "software toy" genres, settings the foundation for things such as SimCity.
- First commercially available PARC/Apple inspired graphical user interface.
- Pinball is a kind of computational machine.
- In addition to machine layout, you can tweak the laws of physics, and map scoring and sound relationships.

Smith et al. (1994) describe it as "programming by direct manipulation"—in the domain of pinball games. "The elements begin functioning as soon as they are dropped into place." The challenge they set out is to "increase the generality without losing the ease of use."

Spreadsheet

First: Visicalc by Software Arts (1983)

The spreadsheet is credited with establishing the personal computer industry, transforming it from a hobbyist pastime to an essential business tool. The spreadsheet has also been an evocative object to "think with" (Turkle 1984) for computer scientists—e.g. Alan Kay (1984a; 1984b) and Terry Winograd (Winograd 1996).

fx =sum(A1:a3)

	A	B
1	10	
2	20	
3	30	
4	=sum(A1:a3)	
5		
6		
7		

60 ×

Google Sheets

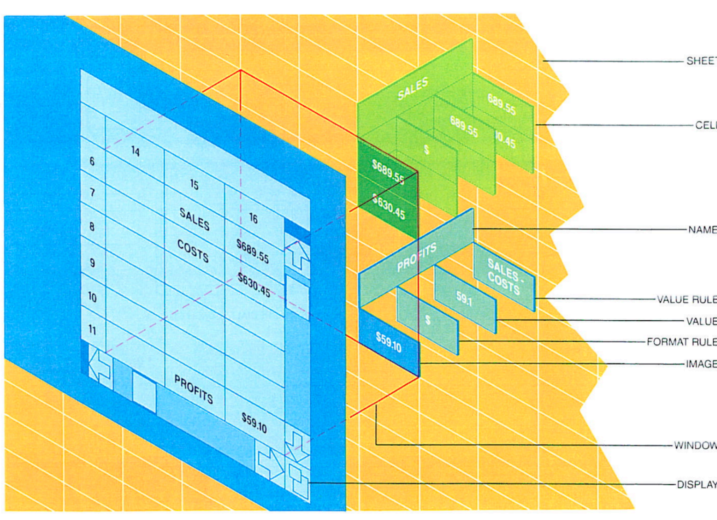
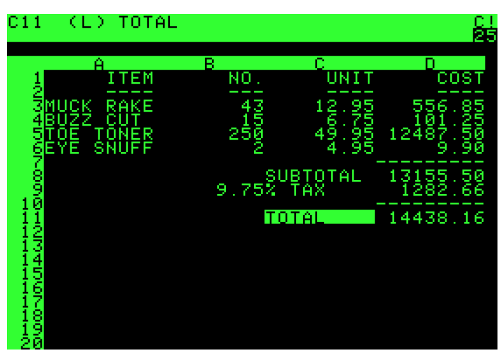


Illustration from Alan Kay (1984) "Computer Software."

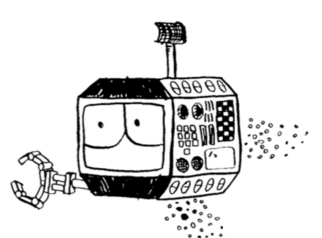


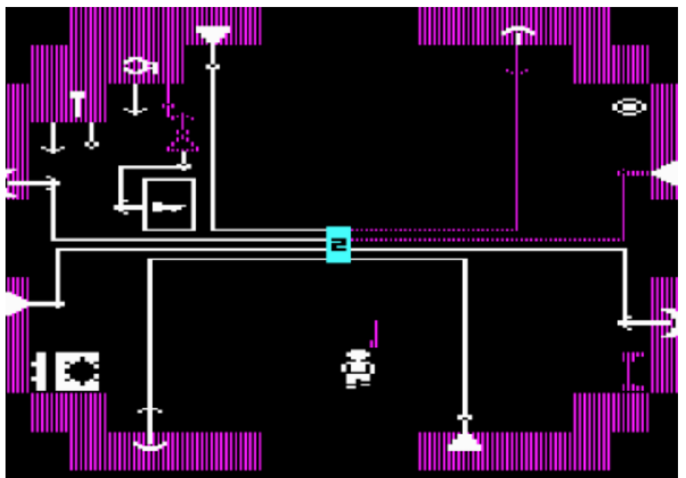
Visicalc on an Apple II

- Data centric. Input/output always visible.
- Organize data in tabular form.
- Cells can only contain formulas, and show the result of those formulas.
- Instant and incremental feedback.
- Spatial. References are spatial.
- Non-spatial references are hard; can't easily refactor functions and variables out of the table.
- Graduated involvement and learning. Begin by reading a sheet, then tinker with data and templates, modify, and finally create your own.
- Bridges program and data, bridging programmer and user (Winograd 1996).
- Introduces a powerful new representation, a "virtuality" (Nelson, Winograd) of a computing data sheet; can be seen as domain specific (Nardi). However, it is a highly abstract abstraction pattern.
- As an externalized shareable cognitive instrument, it engenders fluid sharing and learning, becoming a communal practice (Nardi 1993).

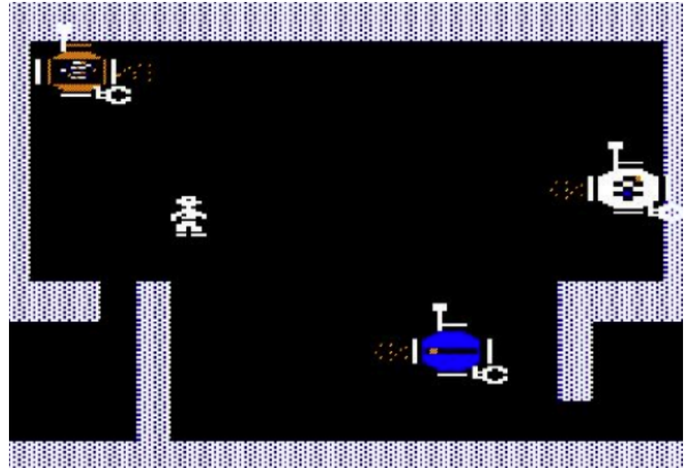
Robot Odyssey 1

Mike Wallace and Leslie Grimm, 1984
Apple II (and other platforms)

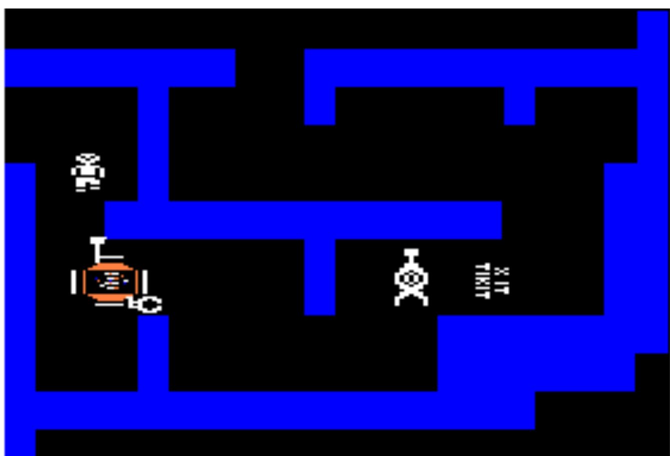


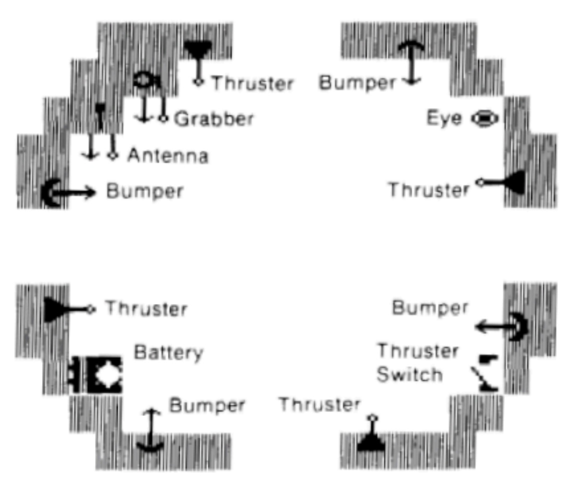


The inside of a robot. Go inside to connect logic to various ports: thrusters, grabbers, battery, eye, antenna, bumpers, etc…



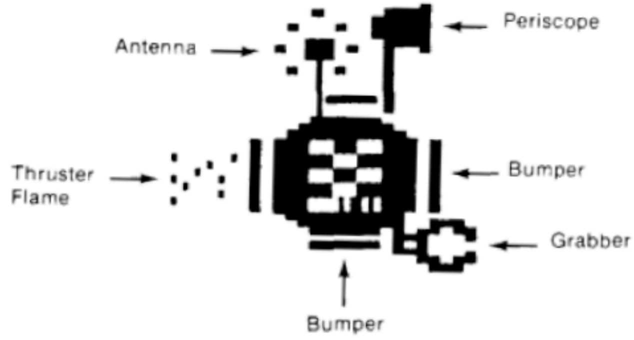
The outside of your robots, which autonomously move around a living robot city. They can be placed inside of one another.





Highlights:

• An elaborate extension of *Rocky's Boots* (Auerbach 2014).
• Extremely difficult. (Intended sequel never made.)
• Adventure game structure. Overcome puzzles that impede your journey.
• World editor (I think)—tools for authoring the world are included and exist inside of the game.
• Fully recursive: design and burn circuits; put robots in one another.



Auerbach, "The Hardest Computer Game of All Time." *Slate*. (2014)

Steamer

Hollan, Hutchins, and Weitzman (1984)

Steamer is an "interactive inspectable simulation" of a steam ship. It is a dynamic and hierarchical "dynamic graphical explanation" of a non-trivial domain—the propulsion system of a Navy ship, modeled in about 100 different diagrams. It also includes an authoring tool for the diagrams.

System overview.

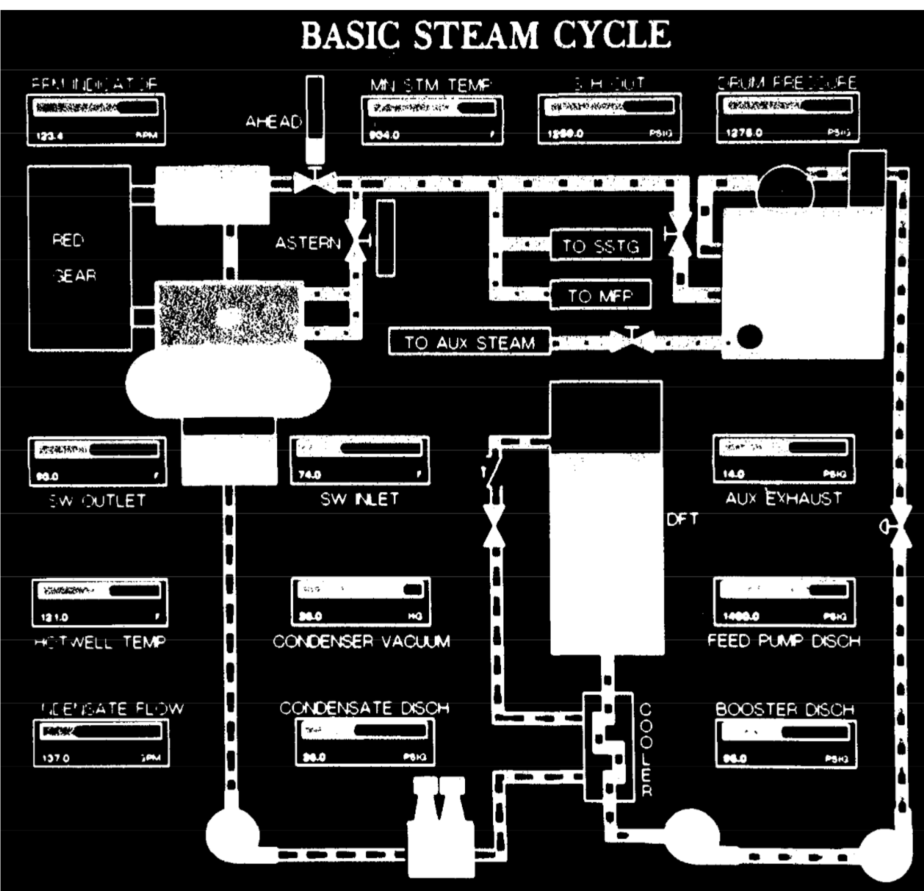
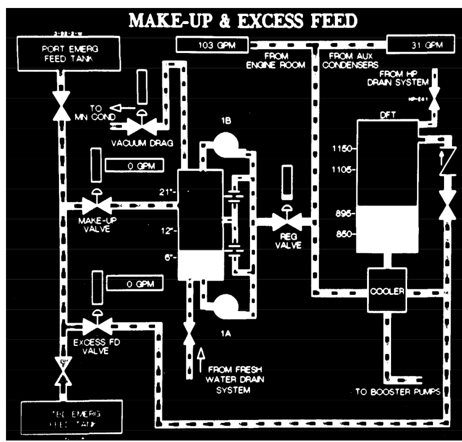
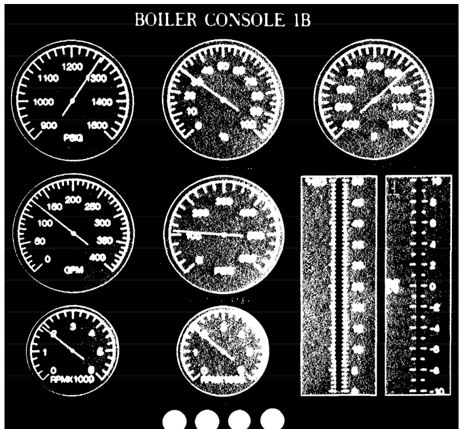


Diagram of subsystem.

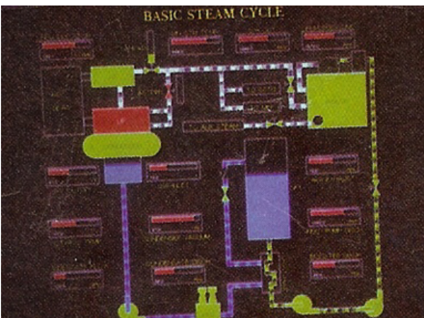




State as traditional gauges

Highlights (paraphrasing the article):

- Allows users to interact with a concrete version of the mental models experts use.
- Internal state can be monitored and manipulated.
- The authoring tool can be used to create "mini-labs".
- "Presenters" discuss how things work, including "procedures," "mappings from abstract abstract generic components and procedures to particular instances."



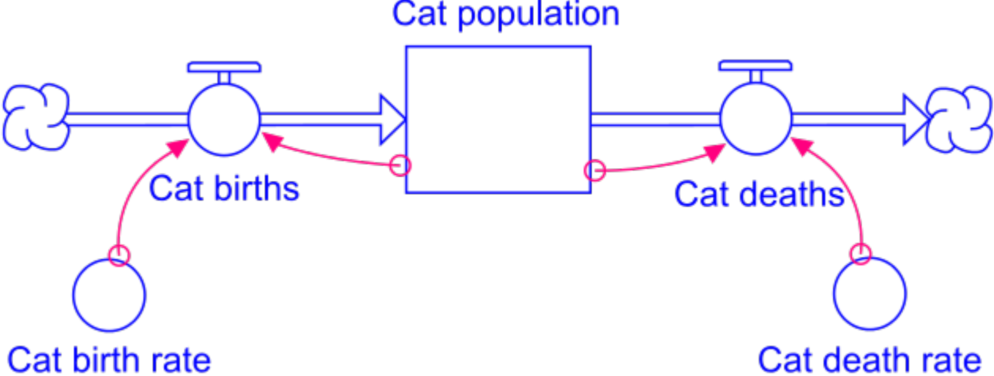
Color images from http://pages.ucsd.edu/~ehutchins/Steamer.html

Hollan, Hutchins, and Weitzman (1984) STEAMER: An Interactive Inspectable Simulation-Based Training System.

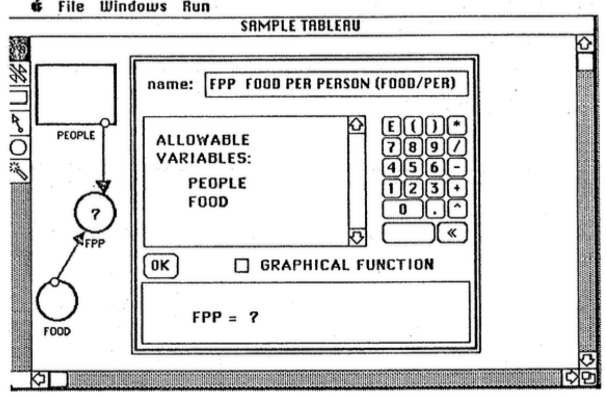
Stella

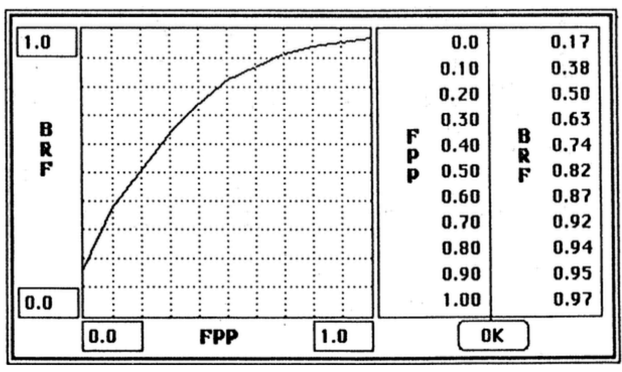
Barry Richmond (1985)

Given how visual and diagrammatic Jay Forrester's notation for system dynamics designs was, it is perhaps unsurprising that an authoring tool like Stella came along. STELLA stands for Structural Thinking Experiental Learning Laboratory with Animation. An explicit goal was to bring System Dynamics to a broader audience by baking in expert knowledge—"model-creation heuristics"—about the modeling domain in order to bring system dynamic model making to a broader audience. Richmond thought that courses and apprenticeships were an inefficient way to spread the practice.



It is interesting to note that Richmond was at Dartmouth, which also the home of another attempt to bring programming to a broad audience: BASIC. In many ways, Stella is the logical successor to Forrester's notation system—carried forth into the era of interactive graphical simulations and direction manipulation interfaces. It's also interesting that it, like Steamer, mark a turn to thinking of expert systems as taking the form of user interfaces—more augmentation than intelligence.



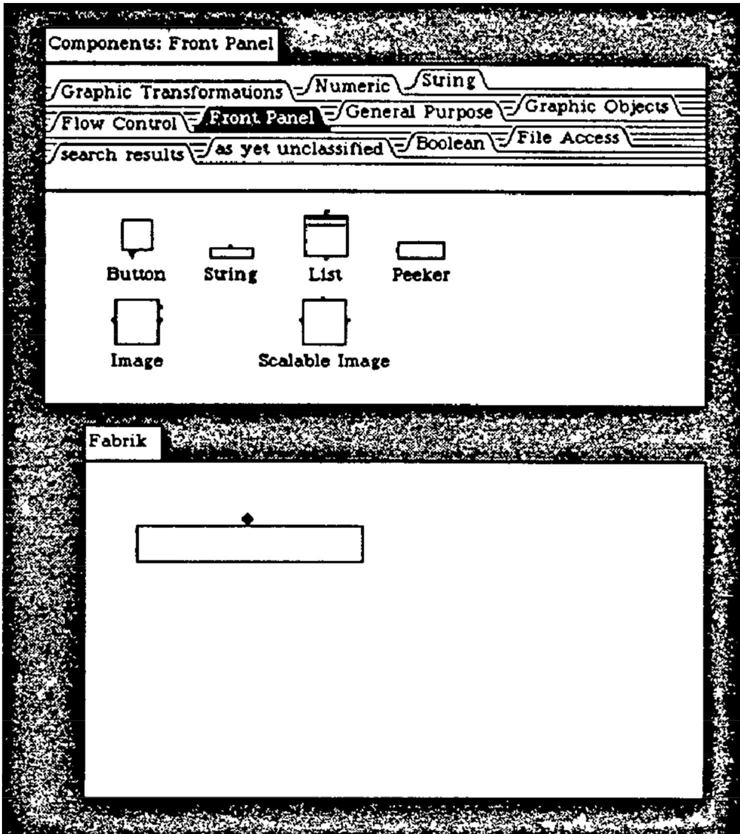


What if, as Alan Kay has suggested in an email to HARC, system dynamics models were animated with particles flowing through the system?

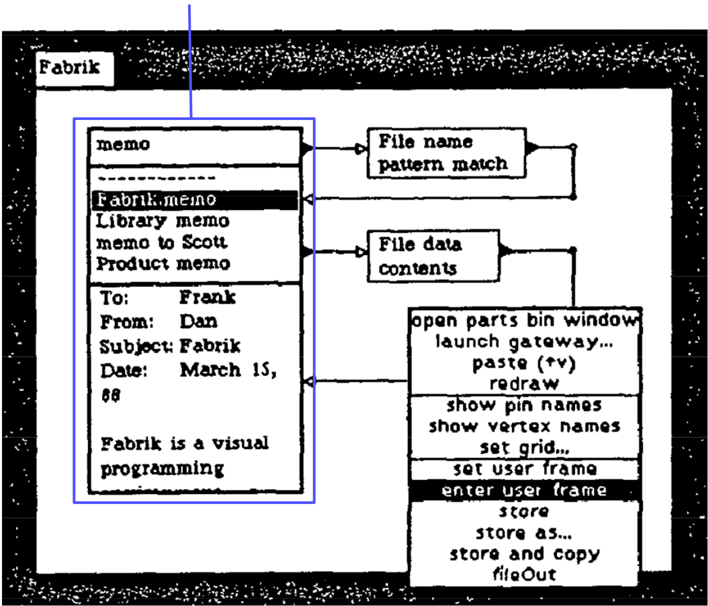
Richmond, Barry. "STELLA: Software for Bringing System Dynamics to the Other 98%." In Proceedings of the 1985 International Conference of the System Dynamics Society: 1985 International System Dynamics Conference, 706–718, 1985.

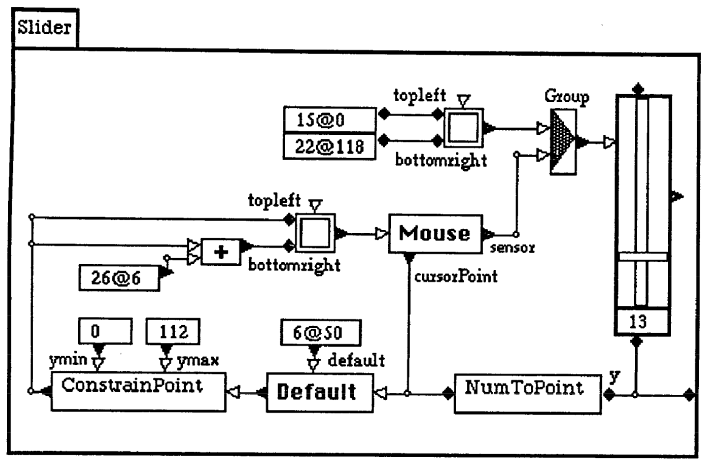
Fabrik: A Visual Programming Environment

Dan Ingalls et al., 1988
Built on Smalltalk (Mac)



User frame



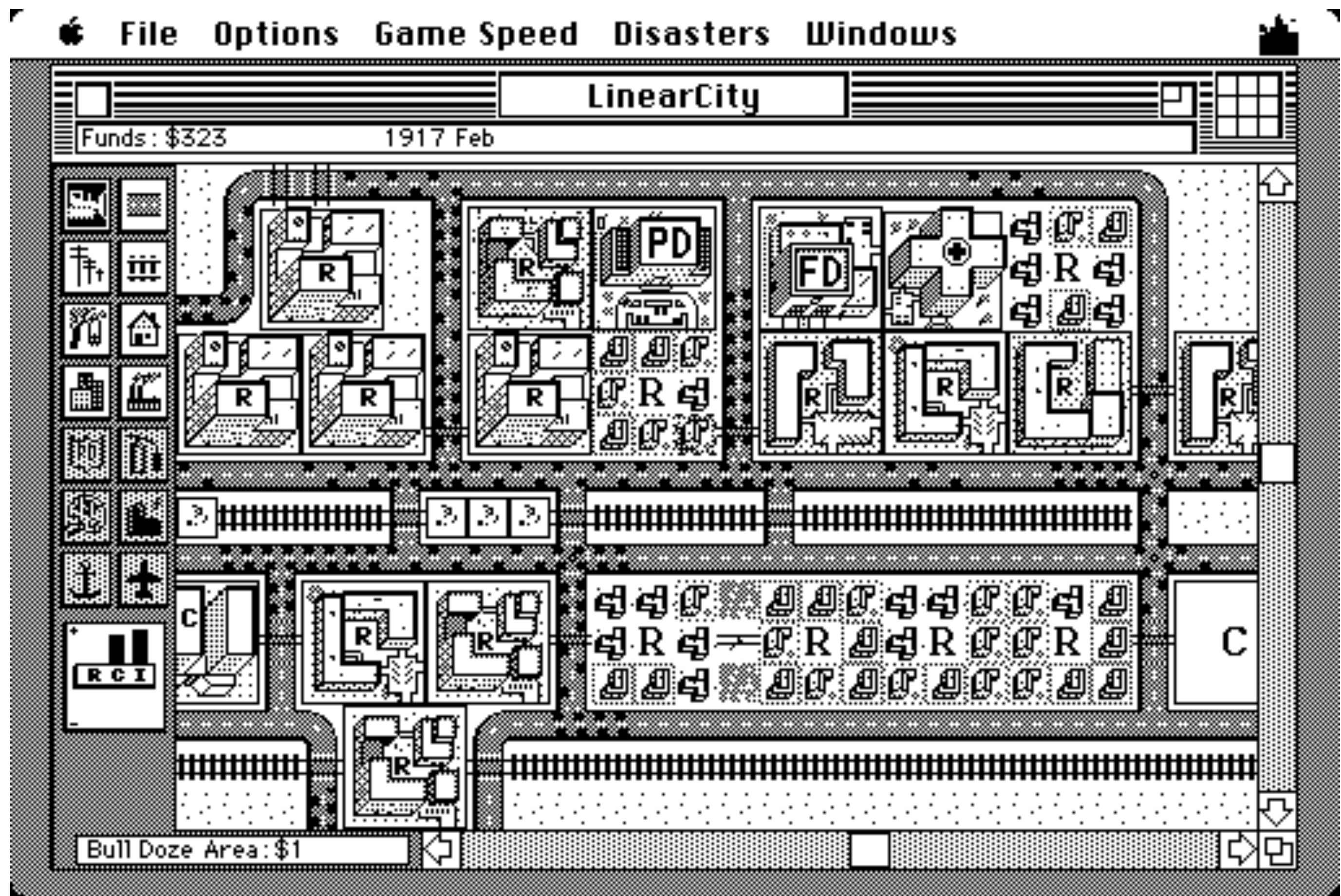


Highlights
- Kit (Primitives, Navigable palette and program, Connectable)
- "Concrete manipulation":
  - Example state always present.
  - Spatial layout and connectivity. ("Visual metaphor" "encompasses" "browsing, testing, connecting" and "using.")
- "User frame"—designate parts of the diagram as external vs. internal.

SimCity

Will Wright et al., Maxis (1989)

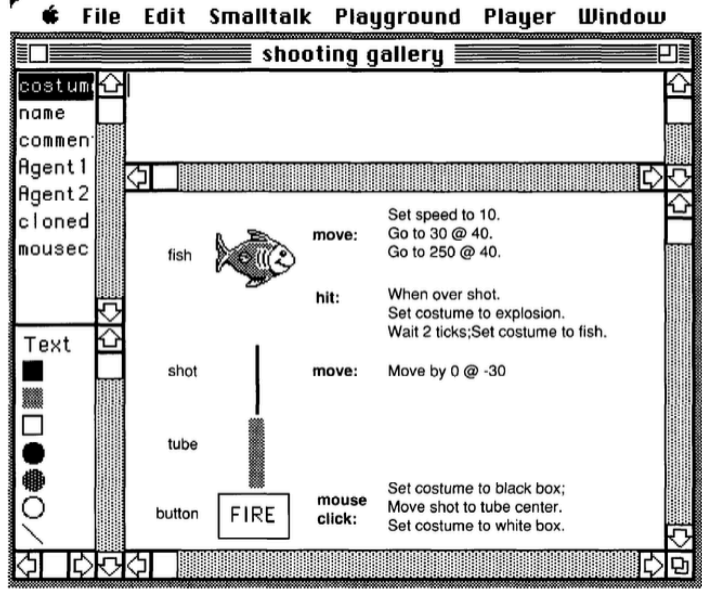
SimCity



Playground

Fenton and Beck (1989), along with Kay, Marion, Beck and Wallace.

Part of the Vivarium project, Playground is an object-oriented language designed for children. The idea was that children would imbue graphical objects with rules, "turning them loose in an envorinment" and thus gain an appreciation for "complex dynamic systems." Particular inspiration is taken from biological systems, which seems to inform many of the examples. Reading Fenton and Beck, the system design sounds like an important historical keyframe between Smalltalk and Squeak, E-Toys, and Scratch.



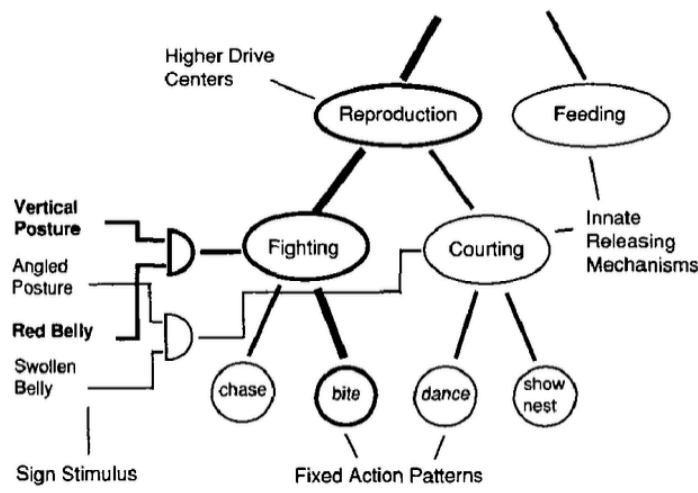
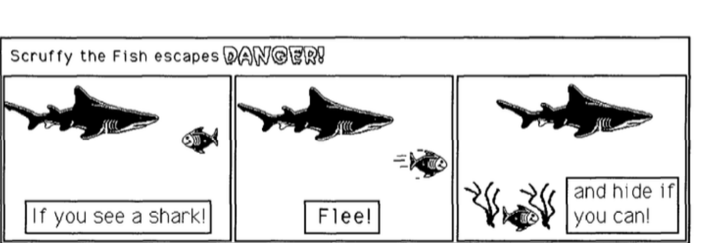


Figure 6: Stickleback Behavior

At first, the program appears to be a drawing program—shape objects, bitmaps, text, and aggregate objects are edited via direct manipulation. This structure is recursive, as objects can be opened up, revealing itself to be another playfield containing agents.

Given their biological impetus, and talk of sign stimulus and drive centers, it's unclear why they didn't offer, as programming representations, (a) behavior trees, and (b) visual diagrams.



Programs are described in terms of causal relations and an "English-like syntax." The authors also speculate that comic book panels could be a good representation for programs.

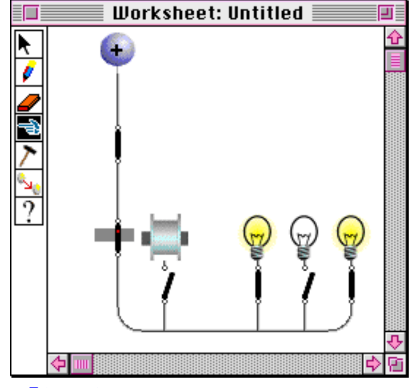
Fenton, Jay, and Kent Beck. "Playground: An Object-Oriented Simulation System with Agent Rules for Children of All Ages." ACM SIGPLAN Notices 24, no. 10 (1989): 123–137.

Kay, Alan. "Computers, Networks and Education." Scientific American 265, no. 3 (1991): 138–148.

Agentsheets

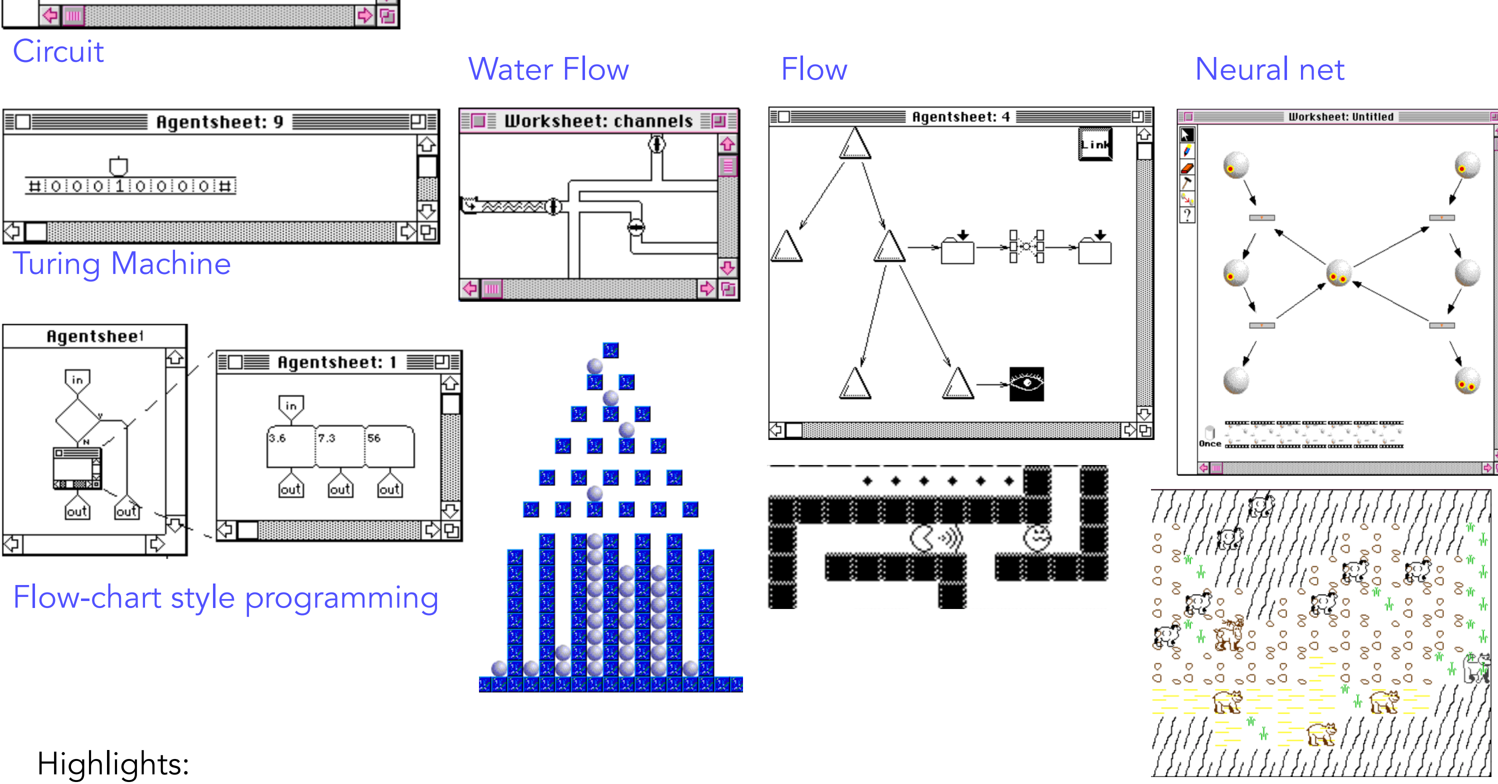
Alexander Repenning (1993–)

Interacting agents are embedded and interact within cellular spaces called sheets. Agents are reactive to direct manipulation and have autonomous behavior.



Circuit

Agent Sheets draws upon a similarly spirited broad field of paradigms: artificial life, visual programming, "programmable drawing tools," "simulation environments", games, cellular automata, and "spreadsheet extensions." Repenning draws upon these shared characteristics: visual, spatial notation, dynamic, direct manipulation, and incremental agency.



Highlights:

- Kits ("agencies") describe specific domains. One effect of "layered" design is "roles"—end-users vs. scenario designers. Example domains in thesis: Turing machines, circuits, flow, traffic, programs.
- Sheet is a cellular 2d space, but agents can be stacked up in a cell.
- Incremental refinement of art, behavior, etc…
- A highly generalize idea of flow is used for things like neural nets, flow charts, water flow, circuits, system dynamic style models, and traffic.
- It also supports ecological style spatial simulations.
- User interaction and agent communication is in the same representation. i.e. Anything can do to one another everything the user can.

The basic tool palette is also a gallery, defined in simulation terms.

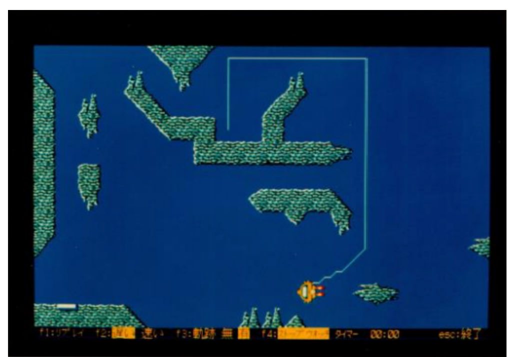
Repenning (1993) Agentsheets. Ph.D. thesis.

AlgoBlock

Hideyuki Suzuki and Hiroshi Kato (1993)

AlgoBlock is designed to facilitate collaborative, socially situated, and meaningful (authentic) learning. They write that "learning is a process of enculturation through social interactions." The authors envision programming languages as "conversational artifacts" that scaffold "interactions among learners."

The primary point of departure from traditional languages—e.g. Logo—is reimagining the screen based user interface, which affords interaction only by a few viewers, with a tangible block interface. The program controls the behavior of an agent, an underwater submarine, in a simulated microworld. Program blocks represent Logo inspired movement commands and control structures. Some blocks have physical switches on them for parameter control.



Ease of use facilitates immersion in group activity rather than the tool itself. Furthermore, it "promotes trail-and-error," which stimulates interaction. Tangibility affords "simultaneous accessing" and "mutual monitoring"—everyone can observe and interact with the shared representation and activity. Tangibility also encourages learners to engage in natural turn taking behavior. Tangibility enables a repertoire of actions and coordinating gestures from the physical world to come into play: reaching out, pointing, looking at, turning towards, holding, etc…

Suzuki and Kato, 1993. AlgoBlock: a Tangible Programming Language - a Tool for Collaborative Learning.

KidSim (later Cocoa, then Stagecast Creator)

Smith, David C., Allen Cypher, and James Spohrer (1994)

In KidSim graphical simulations are created via graphical rewrite rules, which also enables a kind of programming by demonstration.

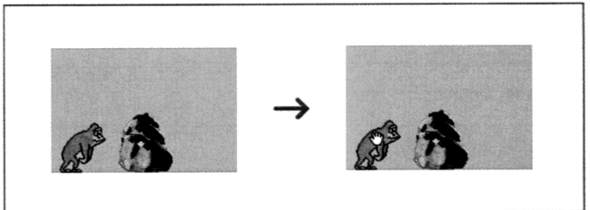


Figure 7. Defining a rule by demonstration

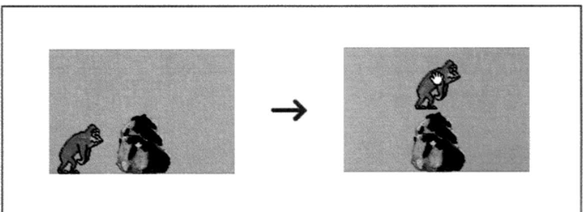
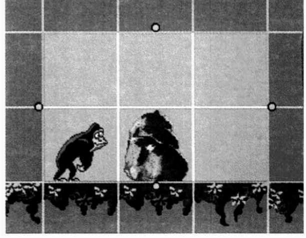


Figure 8. Dragging the monkey above the rock



Specifying the scope of a rewrite rule.

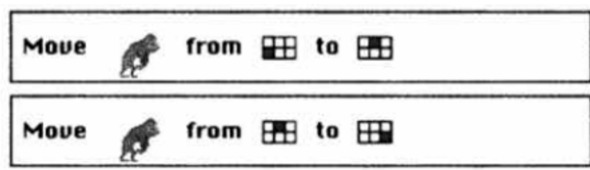
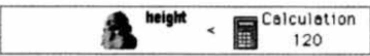


Figure 14. Pictorial display of recorded actions



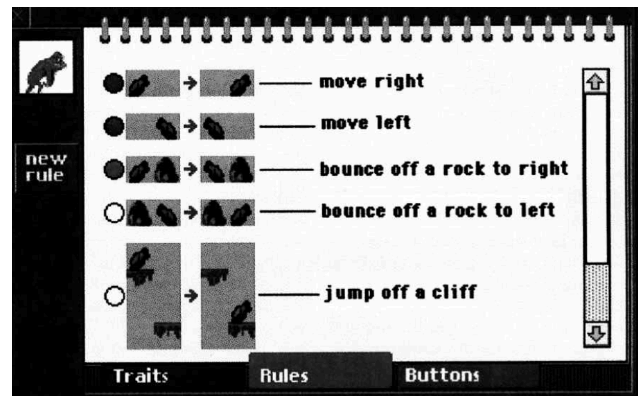
Programming by demonstration extends to using a calculator and dragging properties around to define conditionals.

The creators argue that most people can use editor GUIs (e.g. paint programs), and can give directions, but cannot program. Their solution is to "get rid of the programming language" in favor of a philosophy grounded in GUI design:

- Visibility. Relevant information is visible; causality is clear; modelessness.
- Copy and modify, not make from scratch.
- See and point, not remember and type.
- Concrete, not abstract.
- Familiar conceptual model. ("minimum translation distance").

They choose a symbolic simulation microworld as a domain because it leads to knowing, ownership, and motivation.

All objects are agents which have appearances, properties (name value pairs), and rules.



One of the creators of KidSim, David Smith, was also the creator of another graphical programming environment: Pygmalion.

Smith, David C., Allen Cypher, and James Spohrer (1994)

Visual AgenTalk

Repenning and Ambach (1996)

This paper argues for an augmentation to *visual programming* they call *tactile programming.* The idea here being that program definitions—including program fragments—are dynamic and reactive things that can be run, manipulated, and shared.

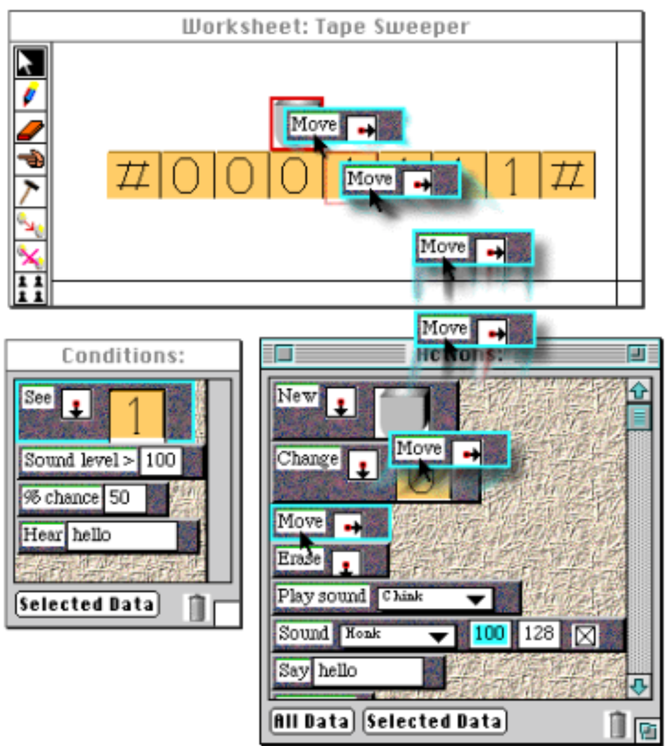
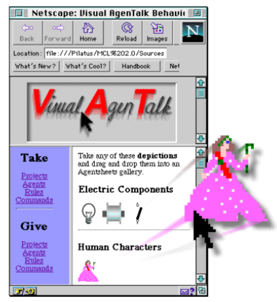


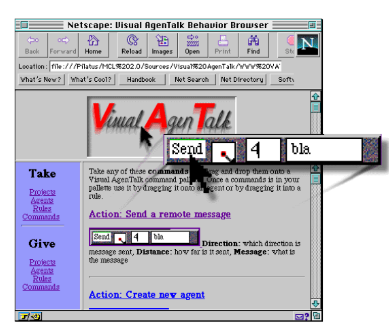
Figure 5: Commands can be dragged and dropped onto agents to explore their functionality and to modify agents

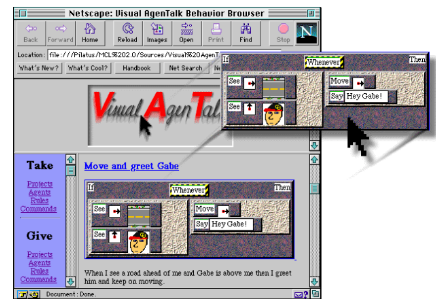
The authors describe tactile programming as "Perception by manipulation". You touch and poke and prod to learn about—not just the simulated world, but the code itself—as well as fragments of that code. In this, there are echoes of Scratch, where you can click on code fragments to run them, and hover over expressions to see their values.

Program fragments can be dropped onto objects. This also has the lovely feel, which they hint at, of lambdas—programs as first class objects that can be sent as messages to other objects.

The social dimension is also very important. Just as Scratch's loose couplings between sprites means greater ease of sharing, AgenTalk seeks to allow sharing at multiple levels of granularity, from entire simulations to components such as objects and code fragments.



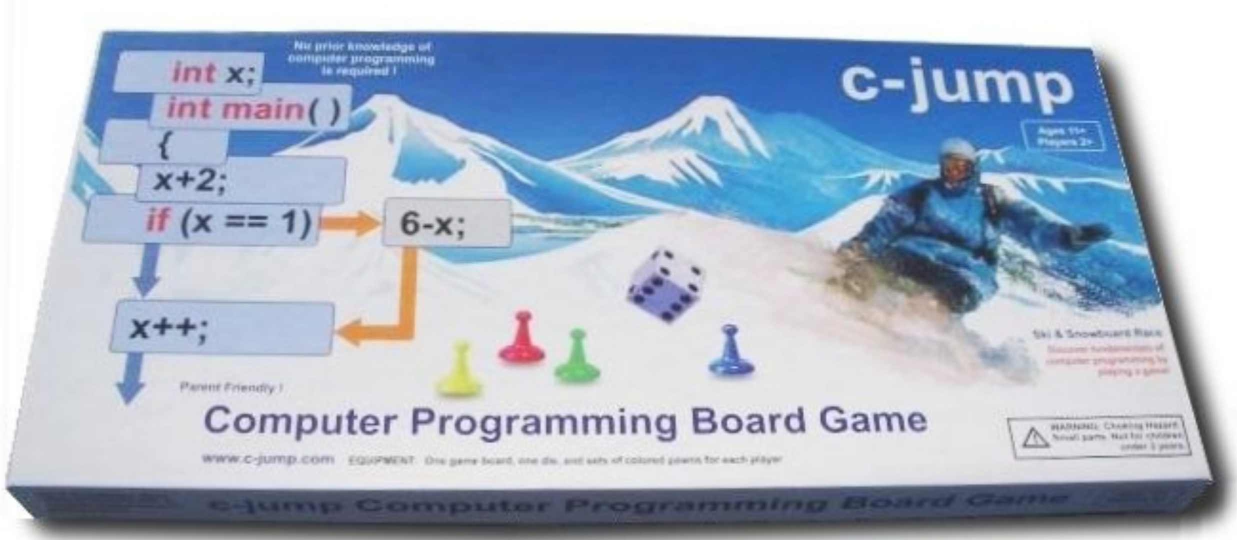




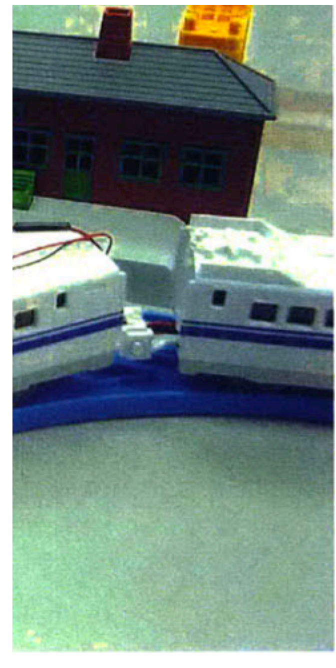
Repenning and Ambach (1996) "Tactile programming"

c-jump

Igor Kholodov (1997)



http://www.c-jump.com



AlgoBlocks (from McNerney 2000); color image from Internet.

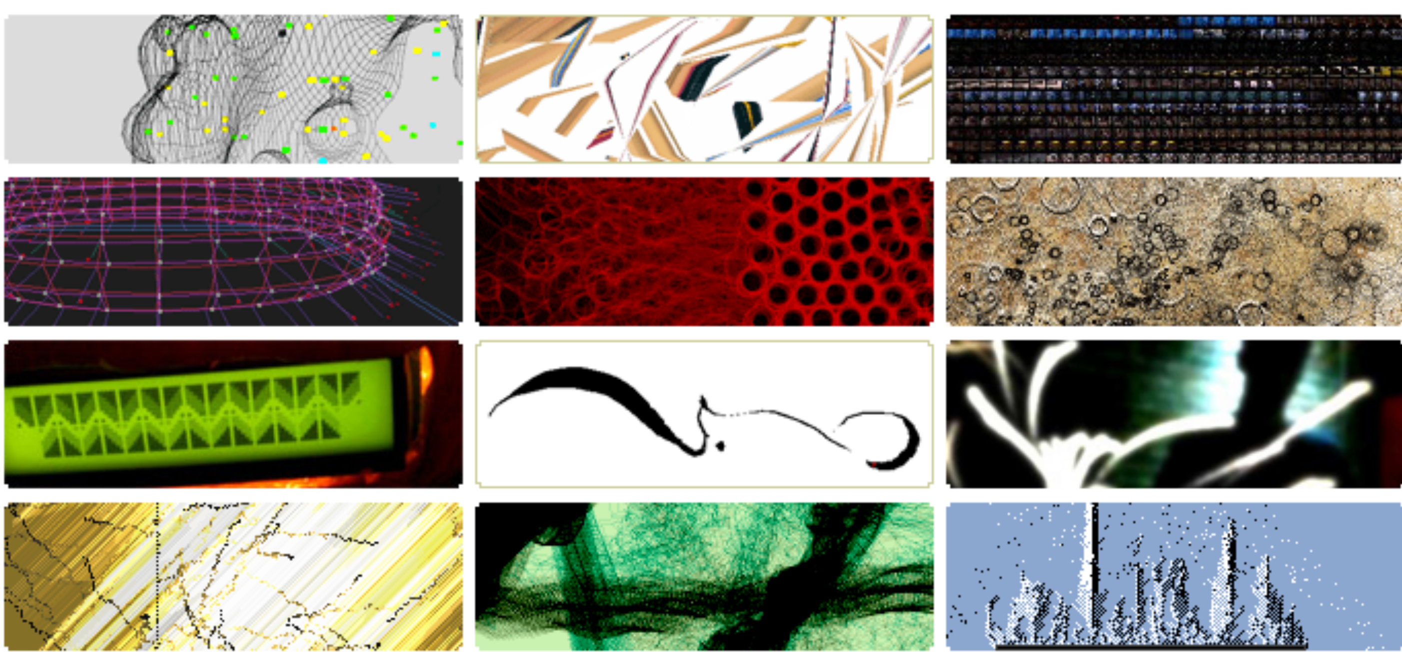
ıgible
n

McNerney 2000

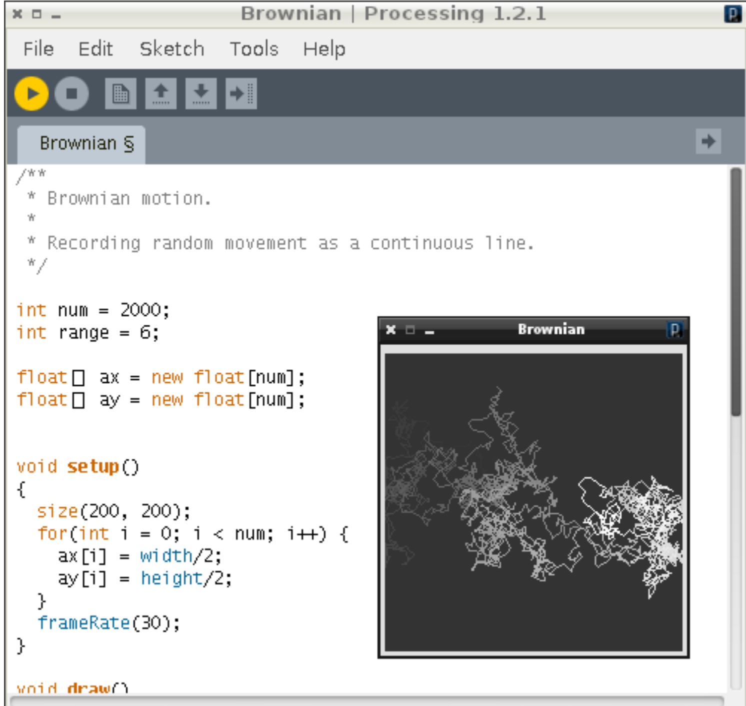
Processing

Ben Fry and Casey Reas (2001)

"Processing is a flexible software sketchbook and a language for learning how to code within the context of the visual arts." —processing.org



Processing is basically Java with a wrapper GUI and a simple and easy to learn set of APIs for drawing, making sound, etc… It has proven to be extremely popular, and now has a Javascript/web incarnation (p5.js).



Why is Processing so successful? Some ideas:

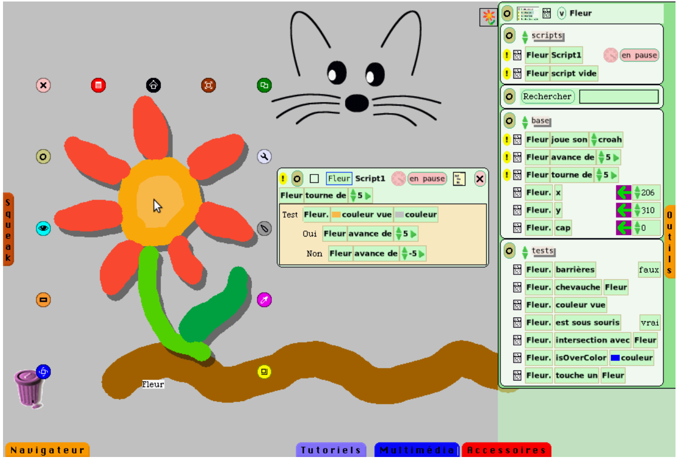
- Targets one domain/community: visual arts.
    - Focuses system design
    - Motivational frame
- Programming is quickly gratifying: art, animation, and interaction!
- Caters to learners and experts.
    - Not a toy environment
    - Easy to play around ("sketchbook")
    - Easy stuff is easy, hard stuff is possible
    - Straightforward examples.
- API exposes key computational and graphical concepts as simple primitives.

Images credits: processing.org gallery and http://www.realtimerendering.com.

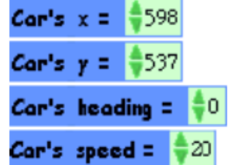
EToys

(1997), built in Squeak

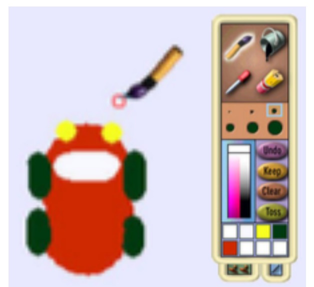
Squeak EToys is an authoring environment for children to think, learn, and create with. It descends from the tradition of Logo, Smalltalk, HyperCard, and emphasizes kinesthetic learning and play as gateways to learning powerful ideas. Scaffolding is intended to be done by teachers, not peers (like Scratch) or games (like Rocky's Boots), and the underlying motivation for engagement is pedagogical, not self-expression (like Scratch).

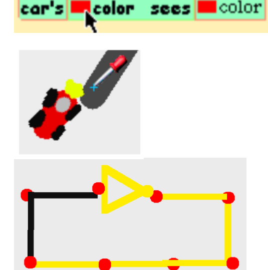


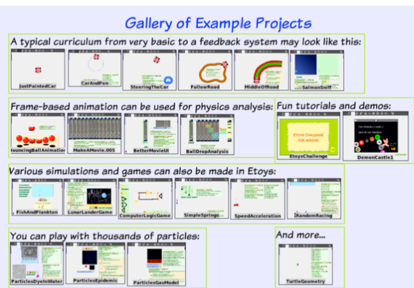
EToys offers inspectors for seeing and modifying things in visual and symbolic forms. The inspectors are quite plastic: variables and code can be dragged into the world. The EToys environment itself can be probed and modified with the same tools.



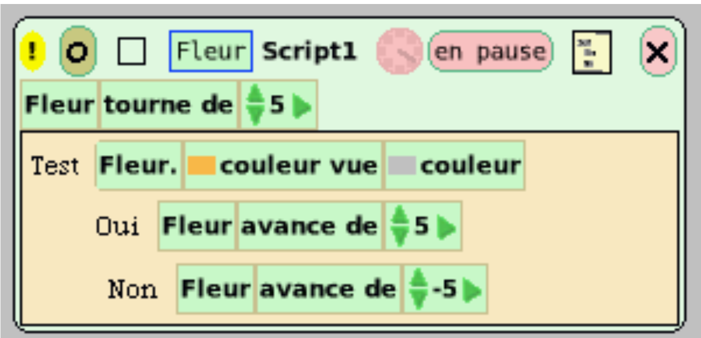
While the interface is powerful, it can be overwhelming and awkward to manipulate. It lacks overviews, and there are multiple spatial/containment hierarchies—the visual world of objects, and an internal hierarchy of objects—to keep clear.







EToys is a spatial, visual world. Objects have locations, sizes, costumes, and headings. They can draw like Logo turtles, and communicate via the framebuffer, exhibiting bitmap costumes and sensing pixels in the framebuffer (SimCity's maps functions similarly). Self-driving cars and Rocky's Boots style simulations are easy to make.



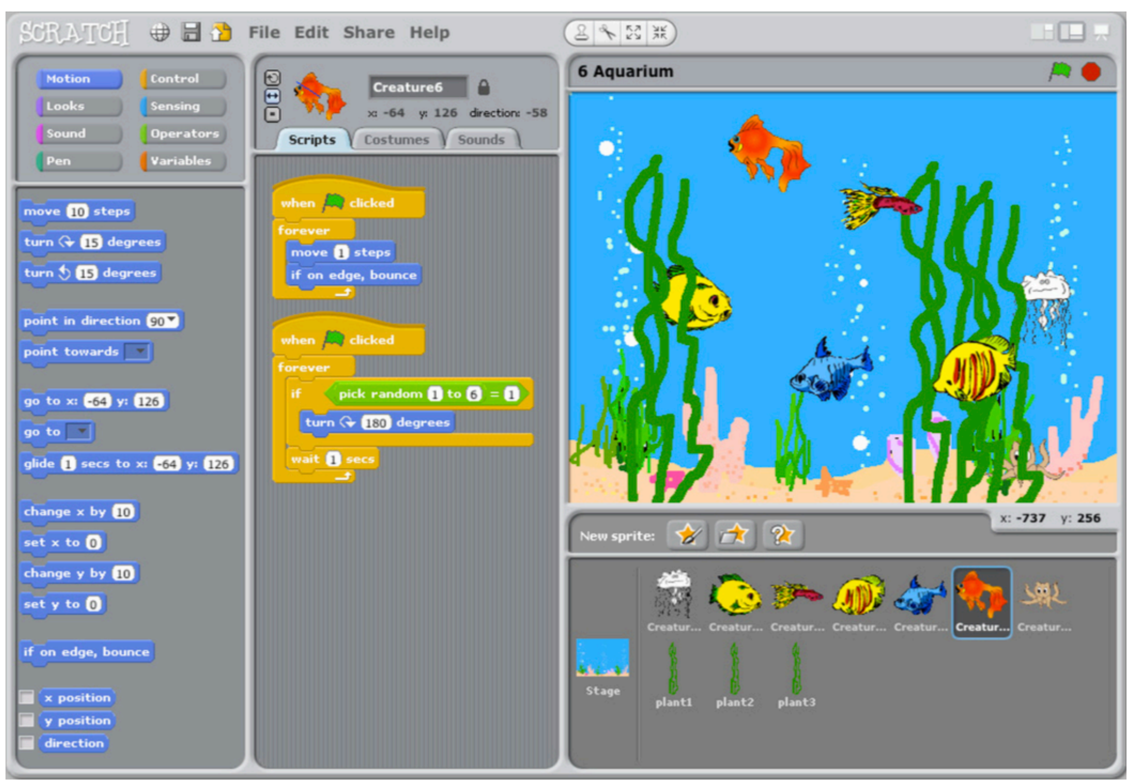
Programming is done with tiles. These are more expressive than Scratch, but manipulation is clumsier. (e.g. attachment combinatorics unclear, and they are hard to disassemble.) Code can live anywhere—in a script, in the world, as a single line or as a script. Clicking "!" executes something, offering a smooth ramp from puppeteering to coding.

Scratch

Lifelong Kindergarten, MIT Media Lab (2004)

Scratch is tool for kids to make "**personally meaningful**" programs like "**animated stories and games**". It supports "**self-directed learning** through **tinkering** and **collaboration with peers**." (Maloney et al. 2004). At http://scratch.mit.edu, users can **browse, play, comment on, see inside**, and **remix** projects.

Tile based programming **Game-like domain**: sprites on a stage.



Overviews: single window; short, browsable command set; color coded command tiles.

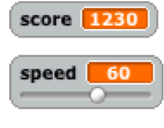
Liveness means code is always running. **Tinkerability** means even code fragments can be run and experimented with—e.g. click any fragment to run it. There are **no error messages**; tile shapes constrains combinations.

Running block outlined in white.

Error shown in red.

running command highlighted in yellow (both when single stepping and running)

expression result shown on hover



Data is concrete—put it on the stage to see it; data changes are animated.

Images from Maloney et al. 2010

MineCraft

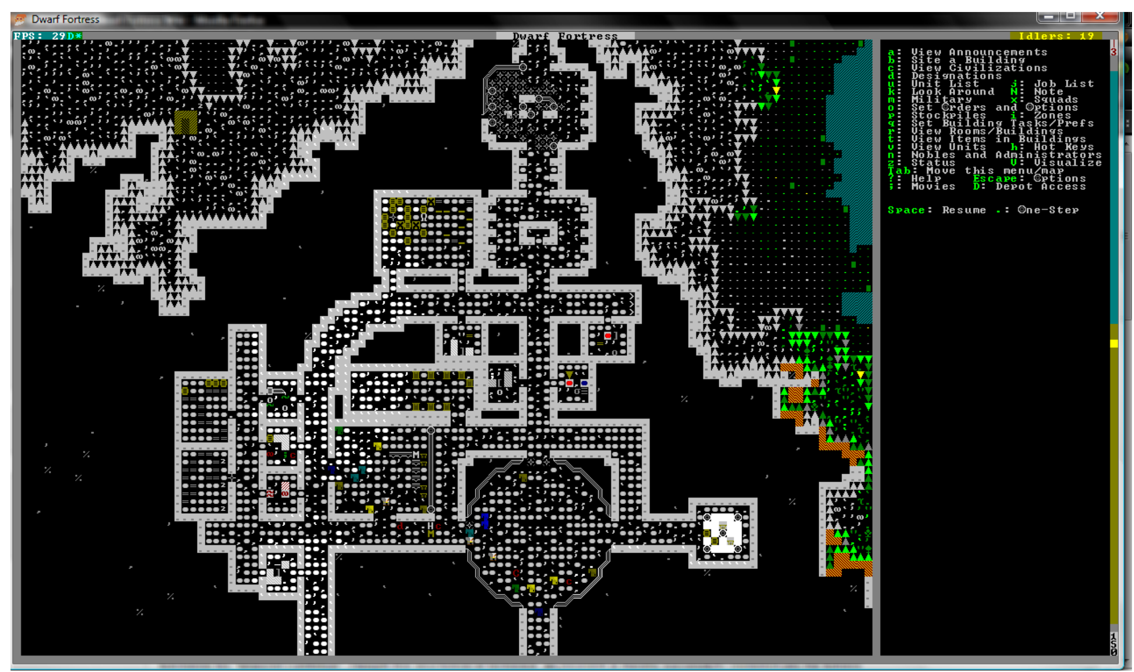
. . . ()

MineCraft

Dwarf Fortress

. . . ()

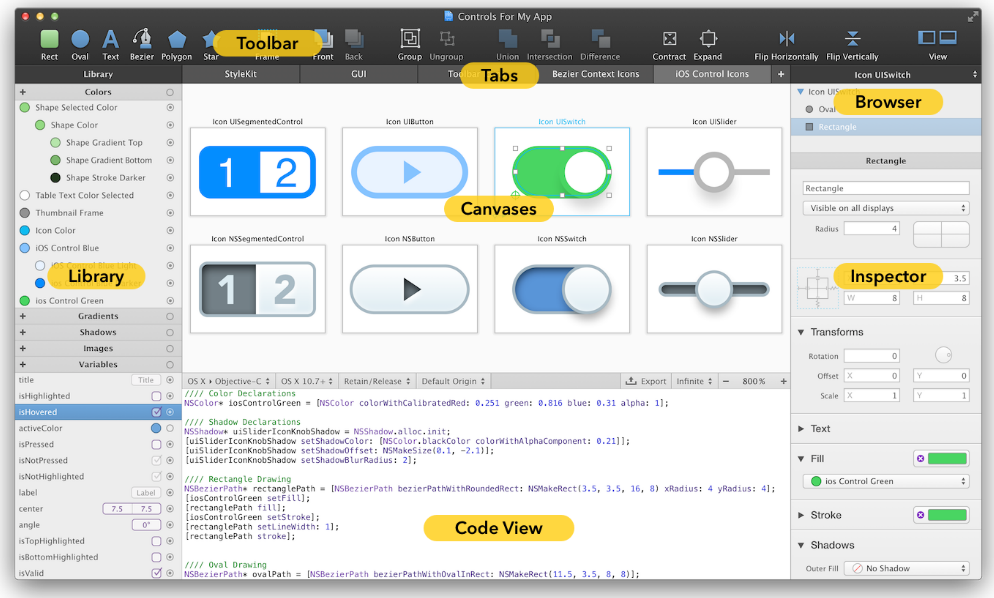
Dwarf Fortress

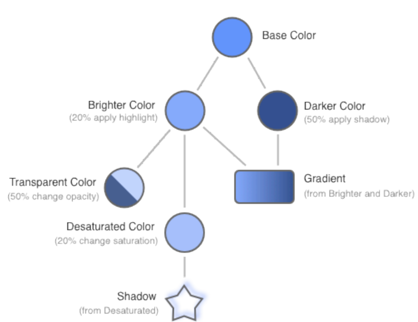


PaintCode

PixelCut (2012)

PaintCode is a production quality illustration program with parametric features that programmers would love. You can design parametric illustrations and then export code (Swift, SVG, CSS) that generates the art with parametric hooks.

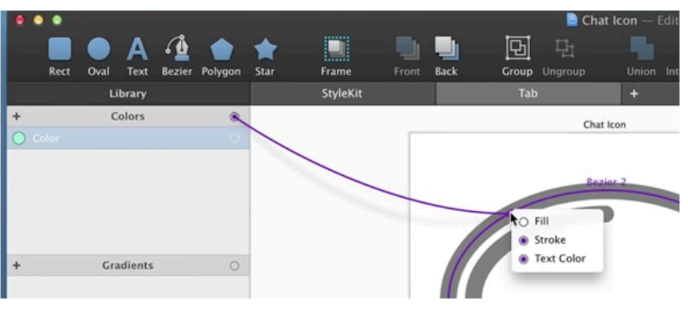


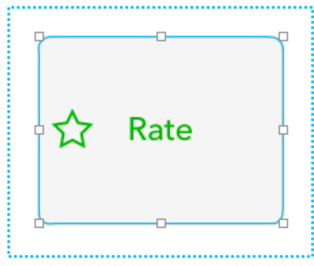


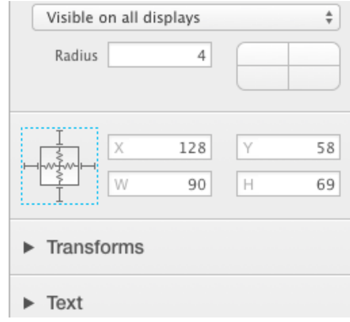
An illustration of how colors are parametrically constrained/generated.

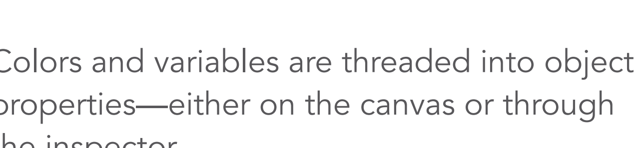
You can't edit the code. The linked representation can help you learn how to program. Just flipping between output languages (Swift, Objective C, etc…) is educational. This is related to the idea of code puppeteering—perform to see the corresponding code generated.

In general, constraints in PaintCode are not systemic (as in Apparatus), but special case per feature; this is how it maintains a production quality user interface that nonetheless captures some powerful parametric features.









Colors and variables are threaded into object properties—either on the canvas or through the inspector.

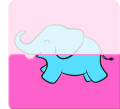
Spatial constraints can be established between special frames and shapes (and vertices), allowing visually adjustable parametric systems to be built up.

Lively Kernel
(2012)

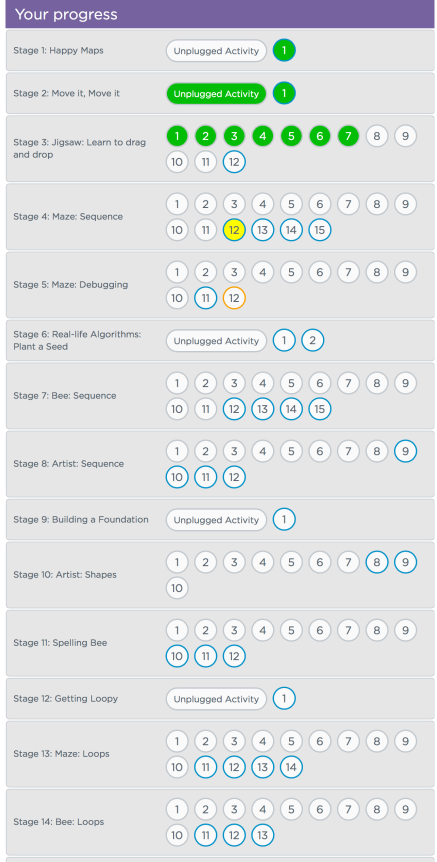
- From SmallTalk: Inline Evaluate (and print) and Run. This idea has been rediscovered over and over (e.g. Light Table).

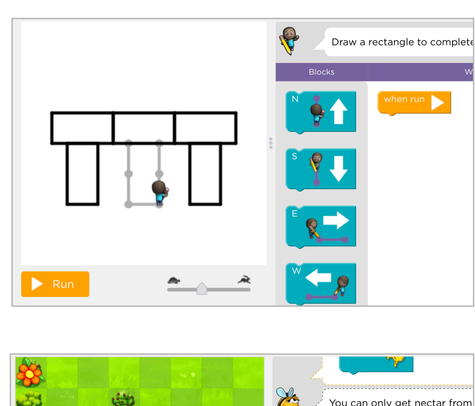
code.org

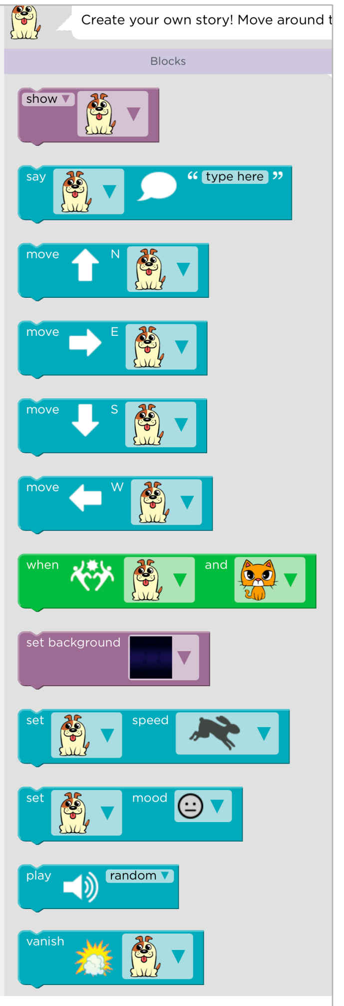
Founded in 2013 by Hadi and Ali Partovi



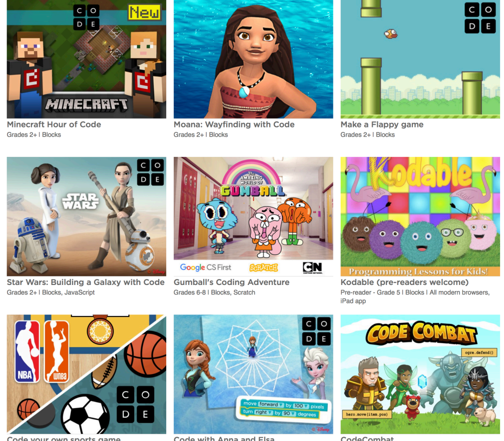
Code.org offers a massive library of programming courses made up of highly accessible—and visually polished—progressions of block based programming puzzles. It accommodates K-12 via courses geared towards specific ages and grade levels. Multiple thematic entry points, including loads of licensed IP, try to accommodate all tastes. The emphasis is on learning to code to solve given puzzles, rather than programming as creative expression (e.g. Scratch).



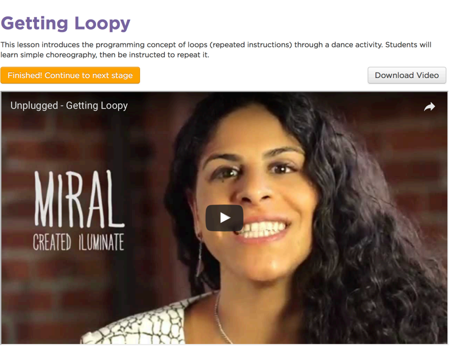




Google's open source Blockly JavaScript library is used.



Themed content.

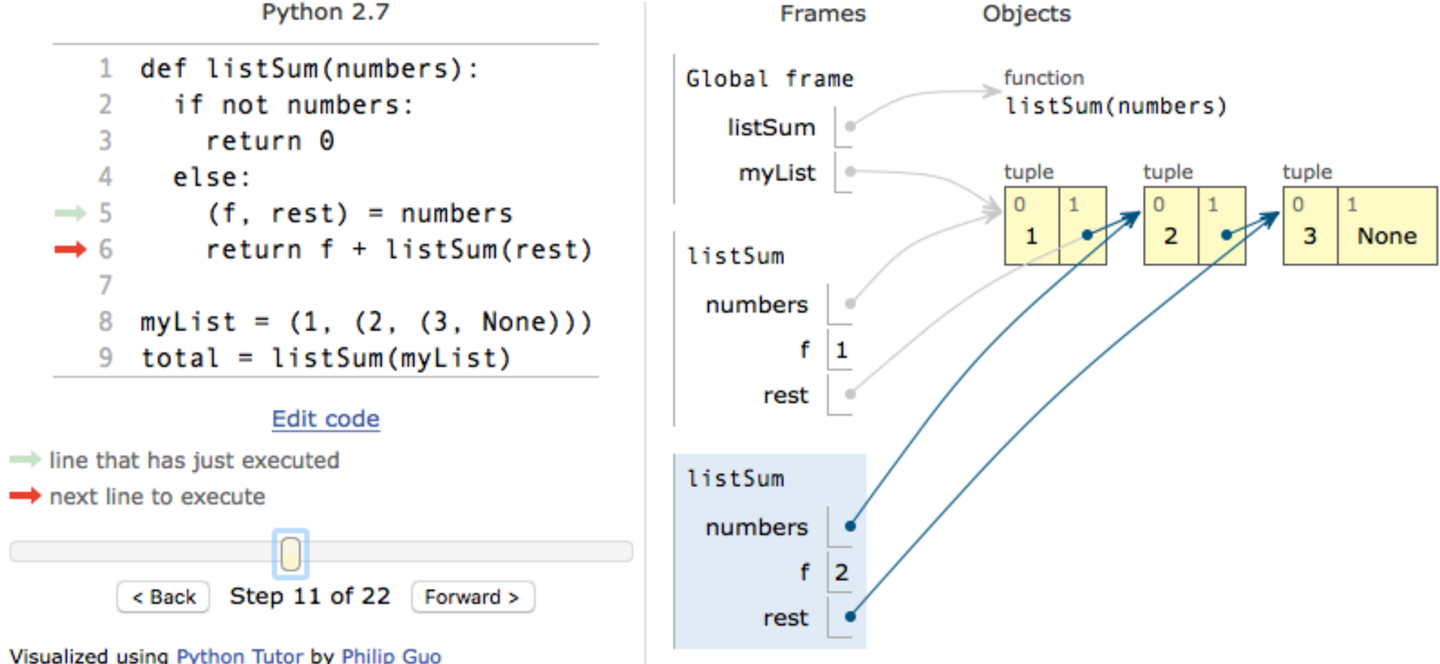


Videos

Python Tutor

Philip Guo (2013)

Visualizes the data structures and execution of programs. Runs on the web, is embeddable, and has achieved a degree of widespread use.



Python Tutor is a popular example of a multi-decade endeavor in computer science: program visualization system for pedagogic ends. It also overlaps with another effort called *software visualization*. Even the Atari 2600 had a BASIC cartridge exhibiting such characteristics, made by Warren Robinett, the creator of Adventure (2600) and Rocky's Boots (Apple II).

Some observations I pulled from surveys by Sorva and others (see references below):

• User motivation and engagement is critical. Sorva et al. (2013) argues that a constructionist orientation is desirable: learners are "makers who want to build things," which "can be harnessed for better learning."

• Level of abstraction of representation is an important choice. Are algorithms or program execution represented? Abstractions chosen reflect the aims of the system builders.

• Emphasis tends to be on generic representations. What if, instead, we allowed that special cased visual designs, perhaps by the programmer, were worthwhile?

For good surveys, see:

• Sorva, Juha. Visual Program Simulation in Introductory Programming Education. Aalto University, 2012.

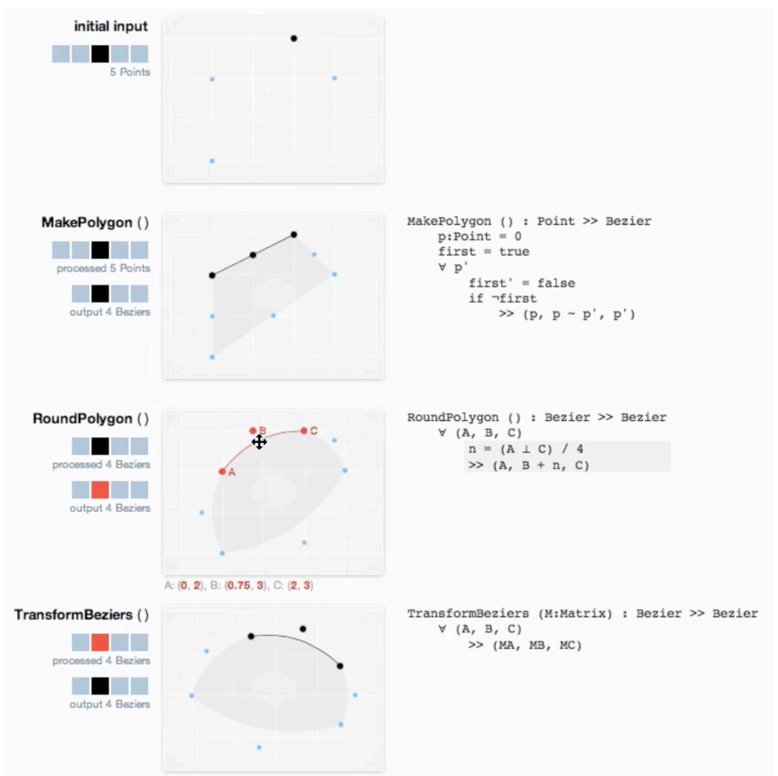
• Sorva, Juha, Ville Karavirta, and Lauri Malmi. "A Review of Generic Program Visualization Systems for Introductory Programming Education." ACM Transactions on Computing Education (TOCE) 13, no. 4 (2013): 15.

See also:

• Guo, Philip J. "Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education." In Proceeding of the 44th ACM Technical Symposium on Computer Science Education, 579–584. ACM, 2013.

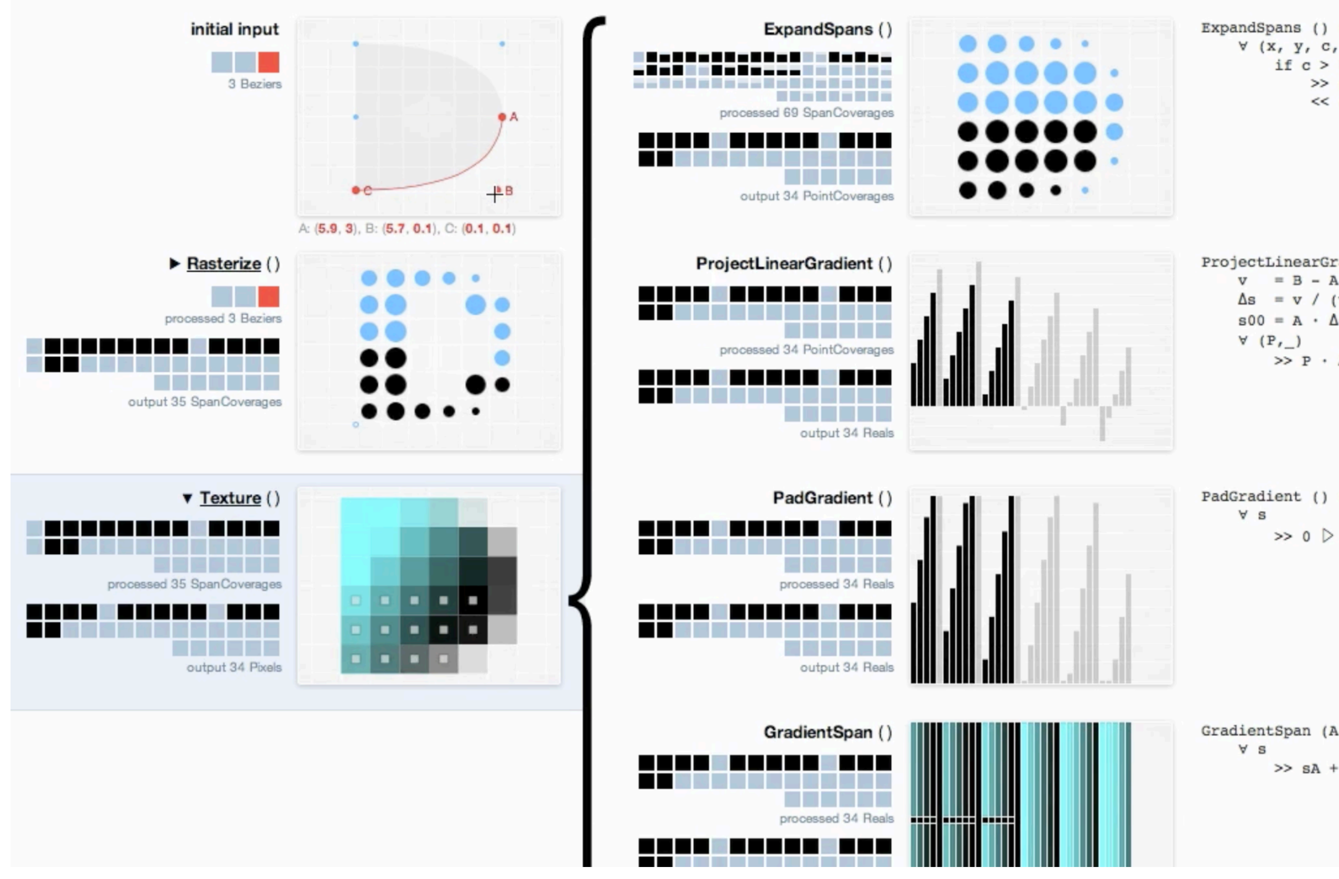
Nile Visualization

Bret Victor (2013)



An interactive visualization of the Gezira/ Nile software by Dan Amelang, a graphics renderer (Gezira) written in a domain specific language (Nile).

This project points to ways in which software description, data, and behavior, can be represented and made tangible as deeply and vivaciously interlinked representations.



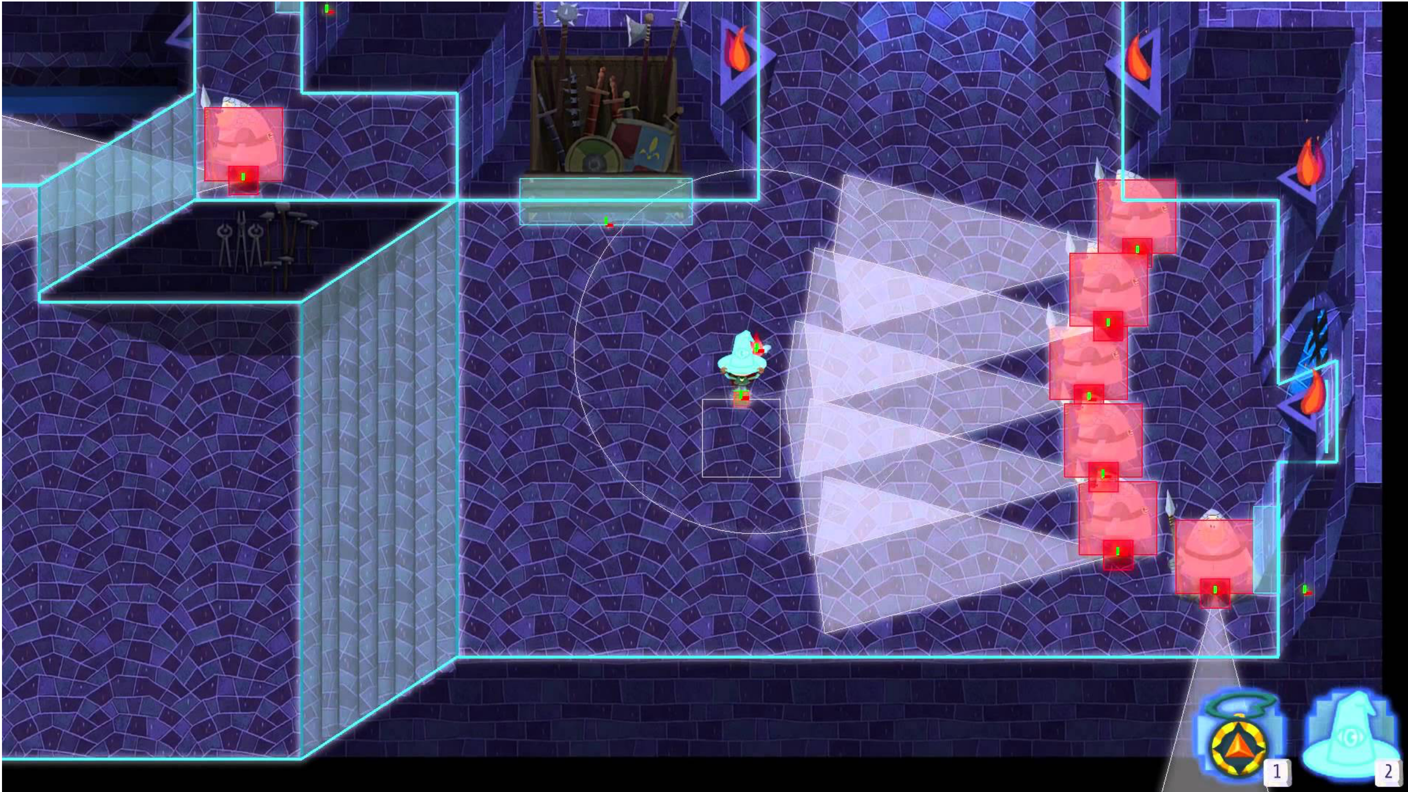
Hack 'n' Slash

Double Fine Productions (2014)

Hack 'n' Slash is a traditional Zelda-like adventure game—an adventure game structure that requires puzzle solving as well as action gameplay to progress—with a twist: instead of a sword, you have a USB stick that can be used to inspect and modify (hack) everything in the world.

The primary mode of interaction is viewing and editing the property sheet of objects.

Below, the player has revealed a visualization of normally invisible spatial data structures:

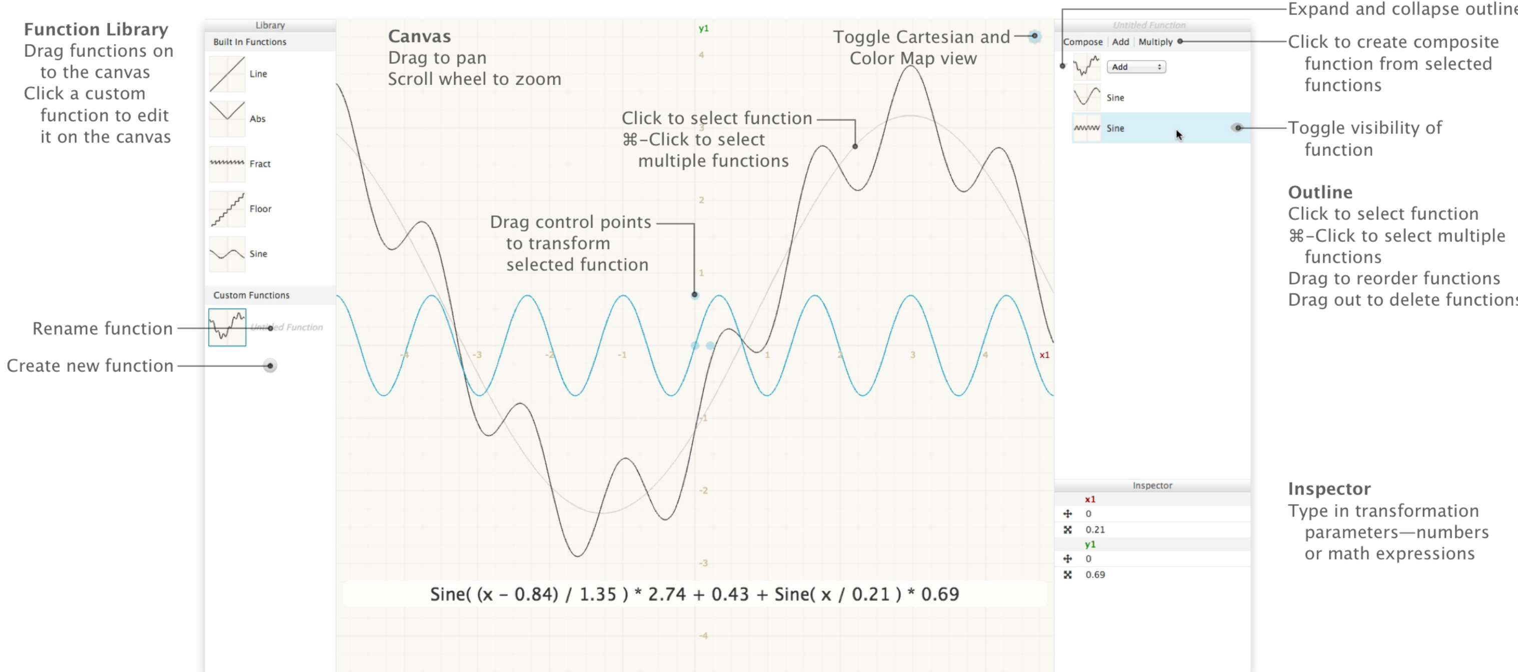


The game works best when it provides an experience that flirts with the boundary between in-game debugging tools that a developer might use, and the experience of a typical genre game. When the game veers off into basic scripting the experience falls apart, as the tools aren't very good.

Shadershop

Toby Schachman (2014)

What if making shaders was more like using Photoshop than writing symbolic code? Shadershop is a direct manipulation interface for constructing shaders out of primitive functions like lines and sine waves, and binary operations like compose, add, and multiply.



Shadershop always shows you the expressed you have created at the bottom, helping you to think across multiple representations. It works in both 1d and 2d domains.

I wish the expression, function hierarchy, and inspector helped me patterns across them more. Perhaps inspector coordinates could be shown in the function hierarchy, and color patterning and pointer interaction could help connect the symbolic expression with the functions. Maybe there is some way to spatially connect the function hierarchy with the expression hierarchy—that would be great.

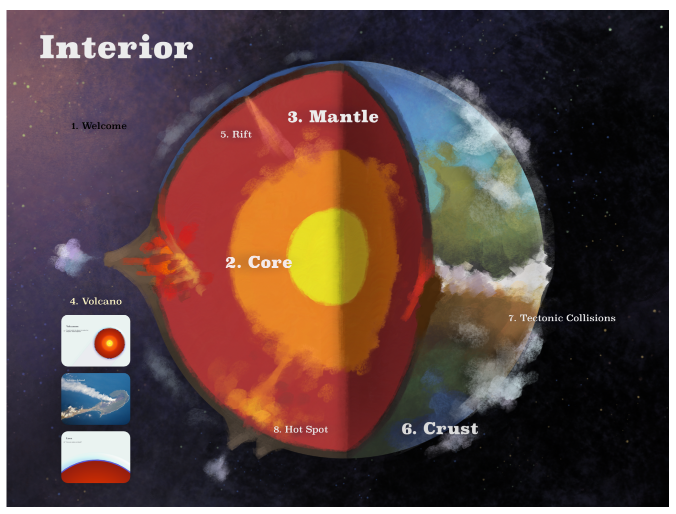
Also, it would be nice if somehow the center of action—the main rendered display—would also function as the primary place where you manipulate all composed functions. Of course this is probably not practical with the 2d view, but it might in 1d.

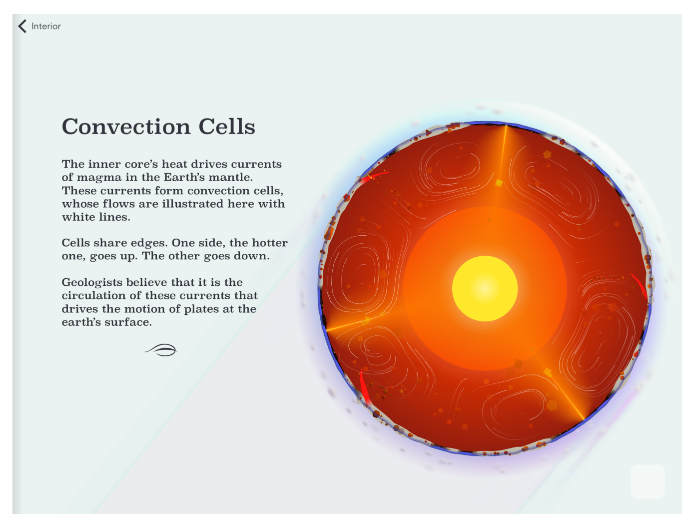
http://tobyschachman.com/Shadershop/

Earth: A Primer

Chaim Gingold (2015)—With Cliff Caruthers (sound), Michelle Lee (illustration), and Laura Kaltman (title).

Earth Primer is a science book illustrated with interactive toys.

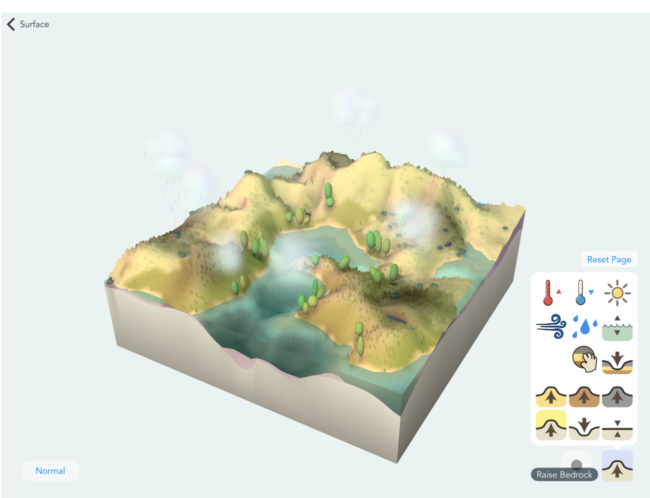


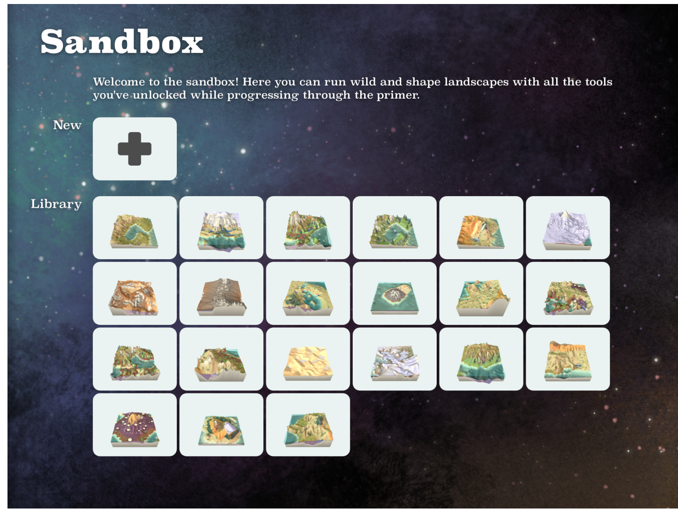


Narrative structure borrows from

- Books: segmentation (chapters, pages), order, and formal genre expectations.
- Games: completing objectives unlocks pages and tools.

Concepts are gradually introduced, creating a fluency gradient in both using the program and geology.





Every aspect of the design is designed to foster delight and wonder: simulation and tool dynamics, interaction design, music, sound, visuals, and animation.

Open-ended simulation play is encouraged in the sandbox mode, and by allowing the narrative intent of most pages to be subverted by open-ended play.

Human Resource Machine

Tomorrow Corporation (2015)

A game about programming. Each stage asks you to write an assembly language program that directs an office worker (you) to manipulate blocks of data.

What works:

- Gentle progression.
- Each level is a simple programming challenge.
- Drag and drop code editor.
- Adjusting playback speed.
- Concreteness: data, program, character taking action, animation.
- Player represented as a character in the program, world, and story.
- Immersive world, story, characters, and music.
- Dialog.
- Optional optimization challenges (# instructions, steps)

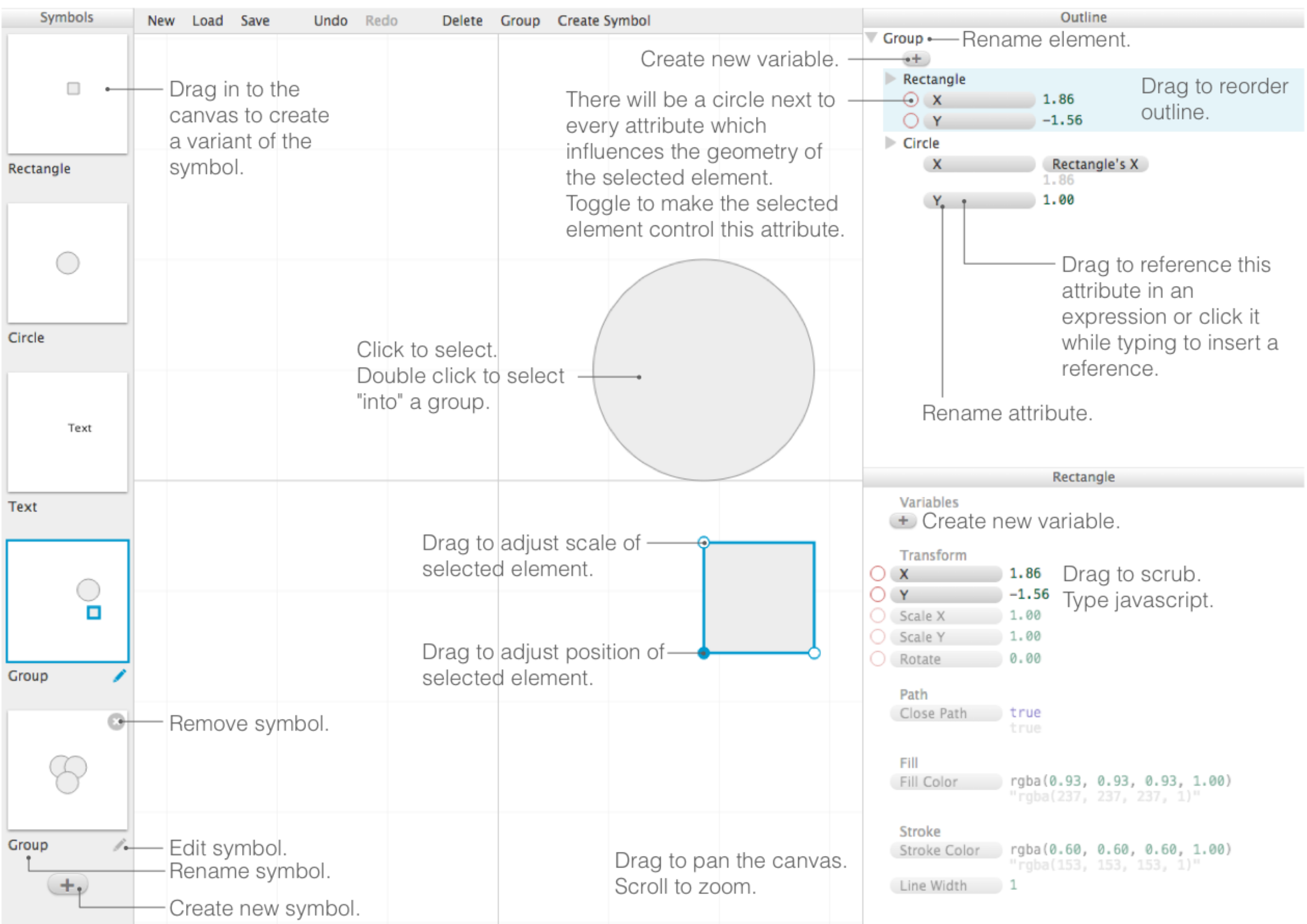
I wish:

- Output was meaningful information, not random data.
- Output could be personally meaningful.
- I could puppet the worker directly to specify action—we could close to this when indicating registers, as we point to the world.
- I could drag program counter.
- More robust time travel debugging. You can only rewind after the program fails. Pause needed.
- World reacts, previews with data and activity, program as I build it.
- Assembly is a liability and an asset.

Apparatus

Toby Schachman (2015), and Joshua Horowitz

"Apparatus is a hybrid graphics editor and programming environment for creating interactive diagrams" (http://aprt.us). It's central fantastic trick is that the diagram elements can be directly manipulated, allowing the diagram and code to be played backwards and forwards. A numerical solver is used to make the magic happen.



What works:

- Powerful causal linkages between visual and parametric elements.
- Solver magically brings these bidirectional linkages to life.
- System has a lot of expressive power.
- Clear linkage between visual and inspector relationships.
- Combination of direct manipulation and coding.

I wish:

- Multiple representations—outline, inspector view, constraints—were more consolidated, ideally as manipulables in the main canvas.
- More responsive performance.
- An ecology of reading and writing:
  - Easy to embed diagrams and make content for them to live in.
  - Easy to share components.
- Complex network of causal and hierarchical relationships was somehow less dizzying.
- Improved graphics tools, e.g. color picker.
- Simulations: feedback loops and time.

g9.js

Guillermo Webster (2016)

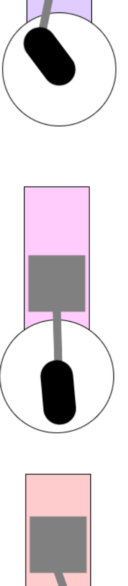
g9 is a Javascript library for making interactive figures. It works like a stripped down version of Apparatus. Figures are described as code, with the variables that move freely with user interaction specifically called out. A numeric solver is used to find new values for these variables in response to dragging of the generated svg graphic.

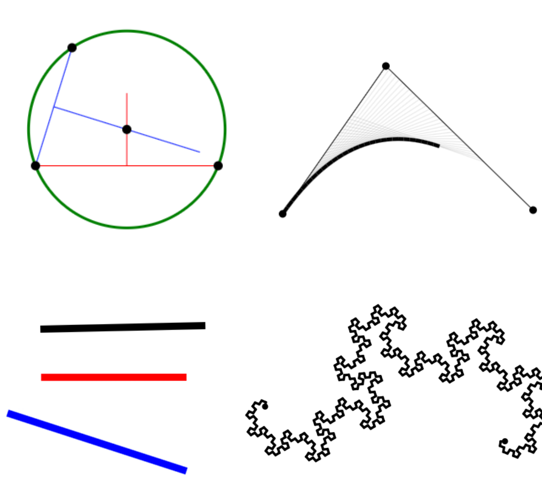
Run (Cmd-Enter)

```
var initialData = {
    x: 10,
    y: 0
}

var render = function(data, ctx){
    ctx.point(data.x, data.y)
    ctx.point(data.y, data.x)
}

g9(initialData, render)
    .align('center', 'center')
    .insertInto('#demo-basic')
```

Run (Cmd-Enter)

```
var initial = {
    "time": 72
}

function render(data, ctx){
    var crank = 40;
    var shaft = 120

    // housing
    var fill = 'hsl('+ (Math.cos(data.time) * 60 + 300) +', 100%, 90%)'
    ctx.rect(-40, 0, 80, -235, { fill: fill, stroke: 'black' })
    ctx.circle(0, 0, { r: 70, fill: 'white', stroke: 'black' })


    var CrankAngle = Math.asin(crank * Math.cos(data.time) / shaft)
    var OppositeAngle = Math.PI - (CrankAngle + data.time);
    var Y = shaft * Math.cos(OppositeAngle) / Math.cos(data.time)

    // shaft
    ctx.line(Math.cos(data.time) * crank, Math.sin(data.time) * crank, 0, Y, {
        'stroke-width': 10,
        'stroke': 'gray',
        'stroke-linecap': 'round'
    })


    // crank
    ctx.line(0, 0, Math.cos(data.time) * crank, Math.sin(data.time) * crank, {
        'stroke-width': 40,
        'stroke': 'black',
        'stroke-linecap': 'round'
    })


    // piston
    ctx.line(0, Y, 0, Y - 70, {
        'stroke-width': 70,
        'stroke': 'gray',
    })
}

var graphics = g9(initial, render)
    .insertInto('#demo-crank')

/*setInterval(function(){
    if(!graphics.isManipulating){
        var data = graphics.getData()
        data.time += 0.05
        graphics.setData(data)
    }
}, 10)*/
```

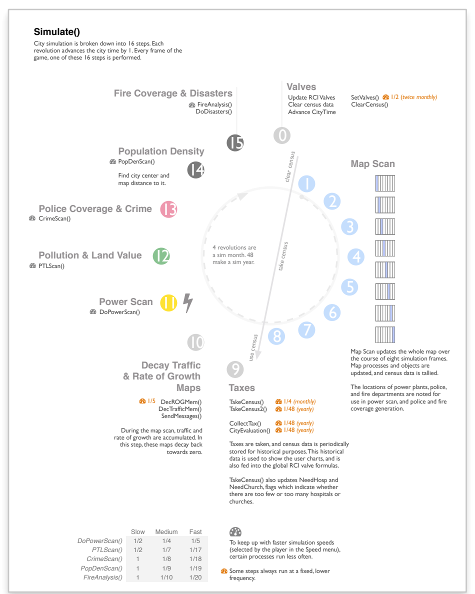


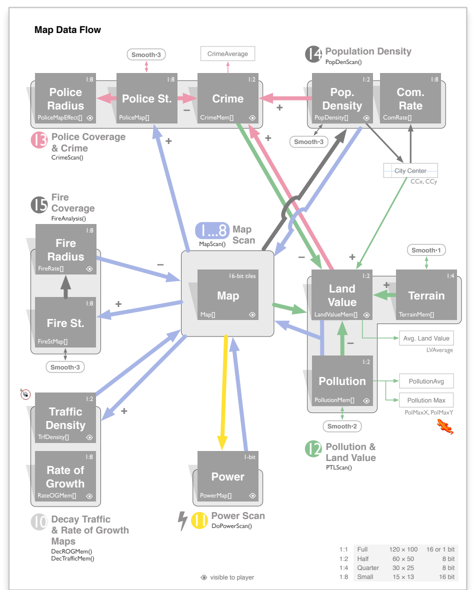


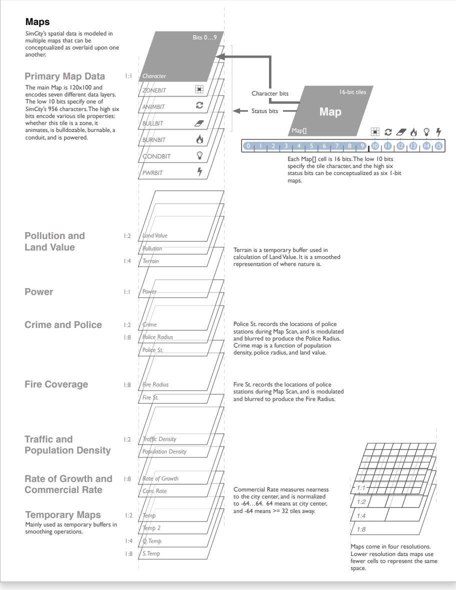
SimCity reverse diagrams

Chaim Gingold (2016)

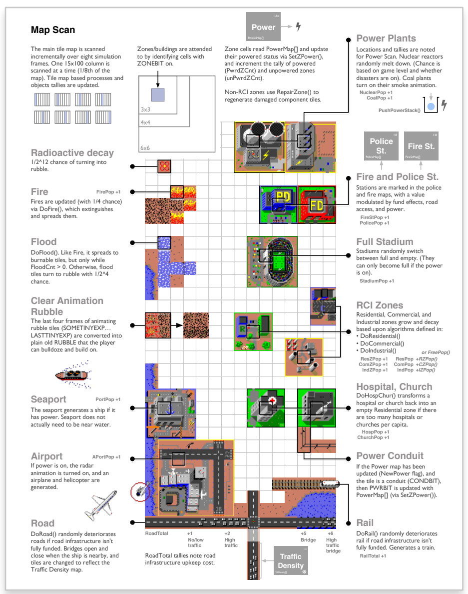
Reverse diagrams (Gingold 2016) map and translate the rules of a complex simulation program into a form that is more easily digested, embedded, disseminated, and and discussed (Latour 1986).

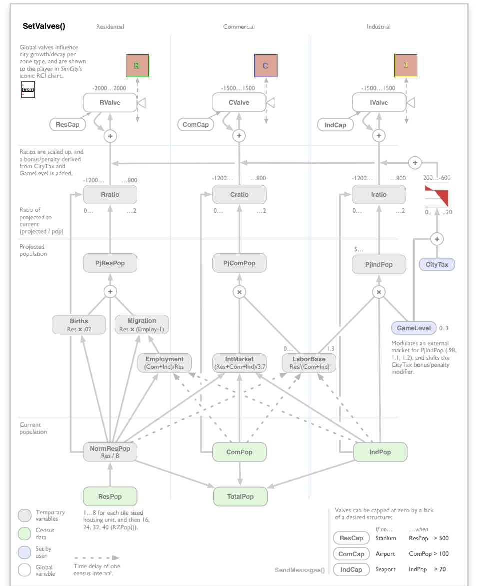


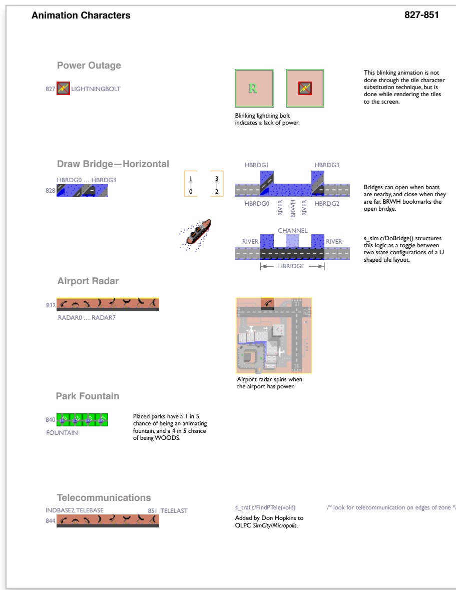




The technique is inspired by the game designer Stone Librande's one page game design documents (Librande 2010).







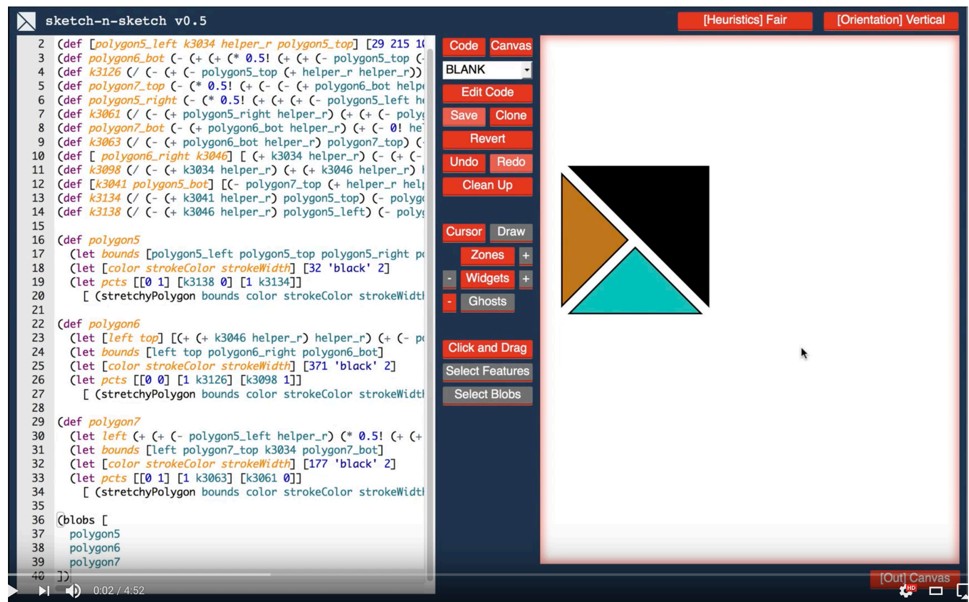
If we merge the reverse diagram with an interactive approach—e.g. Bret Victor's Nile Visualization (Victor 2013), such diagrams could be used ***generatively***, to describe programs, and ***interactively***, to allow rich introspection and manipulation of software.

Latour, Bruno (1986). "Visualization and cognition". In: *Knowledge and Society* 6 (1986), pp. 1– 40.
Librande, Stone (2010). "One-Page Designs". Game Developers Conference. 2010.
Victor, Bret (2013). "Media for Thinking the Unthinkable". MIT Media Lab, Apr. 4, 2013.
Gingold, Chaim (2016). *Play Design*. Ph.D. dissertation.

Sketch-n-sketch

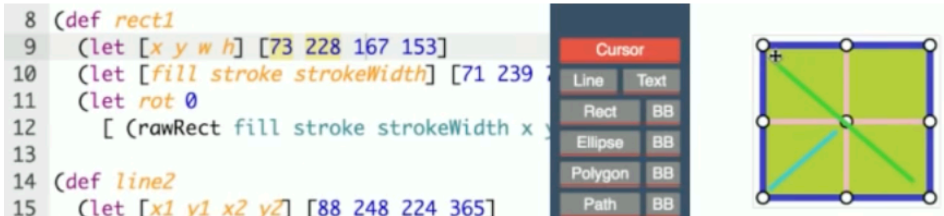
Chugh et al. (2016)

Sketch-n-sketch is an editor for live-linked code and graphics. Edit the code and the diagram changes; edit the diagram and the code responds. It is inspired, in part, by Bret Victor's *Drawing Dynamic Visualizations* and Toby Schachman's *Apparatus.* The point of departure is making a fully featured programming language a top level goal.



I find this to be a very cool idea with some nice flourishes—and I'm sure there is much more to admire. In practice I find it to be quite cumbersome. You have to manually push a "Run" button to get the diagram to respond to code changes, which often leads to syntax errors blowing up the diagram—making the code feel fragile and not incrementally tweak-able, which is a basic asset of direct manipulation. The graphics editor is clunky, and lacks rudimentary user interface niceties. (e.g. I couldn't push delete to remove a selected object.)

One of my favorite touches is that rolling over graphics elements highlights corresponding elements in code.



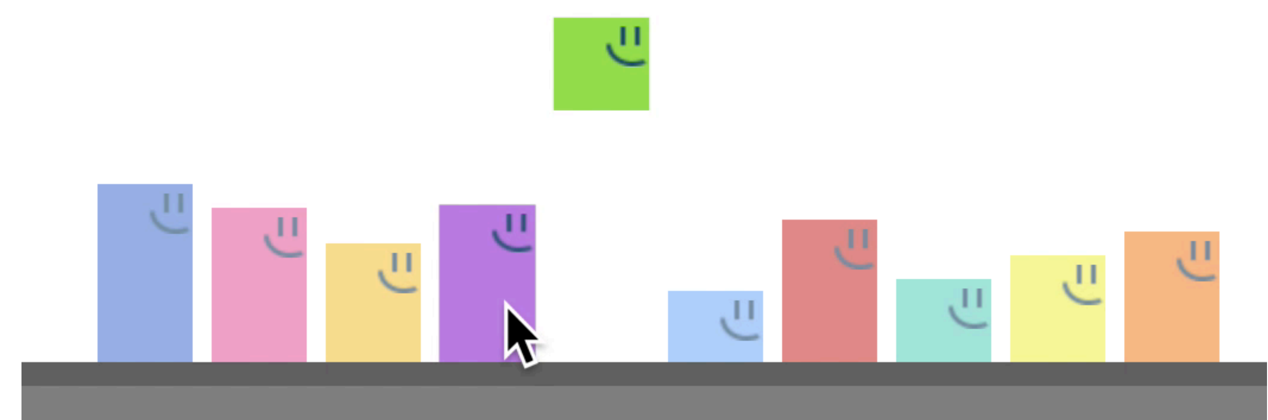
Chugh, Ravi. "Prodirect Manipulation: Bidirectional Programming for the Masses." In Proceedings of the 38th International Conference on Software Engineering Companion, 781–784. ACM, 2016.

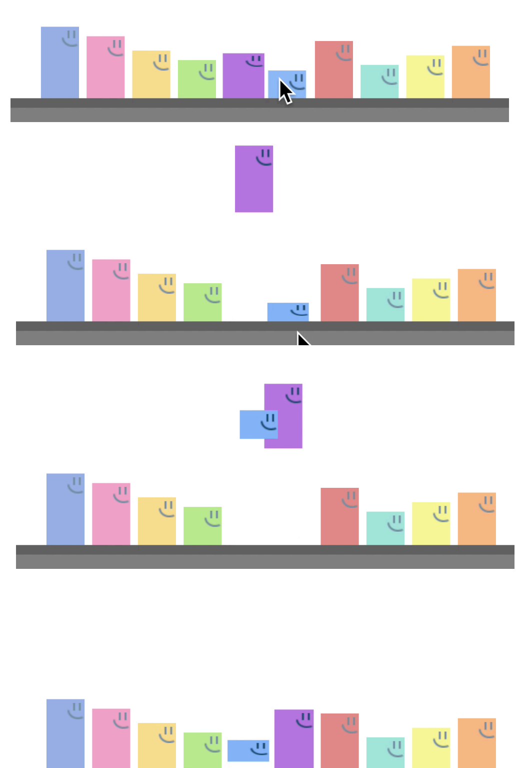
Chugh, Ravi, Brian Hempel, Mitchell Spradlin, and Jacob Albers. "Programmatic and Direct Manipulation, Together at Last." In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, 341–354. ACM, 2016.

Bubble Sort

Glen Chiacchieri (2016)

An explanation of bubble sort that uses "showing and doing", "play", and "exploration." While Glen's piece is a work in progress of a much longer explanatory essay, I want to call attention here to a key interactive:





A line of characters represents an array of data to be sorted. If you touch a character, it jumps—and will jump over its neighbor if they are in the wrong order.

Admirable qualities to note:
• Player puppet the algorithm, from the algorithm's points of view—where it is iterating from.
• Data is concrete.
• Representation is appealing: characters, colors, sound, animation.

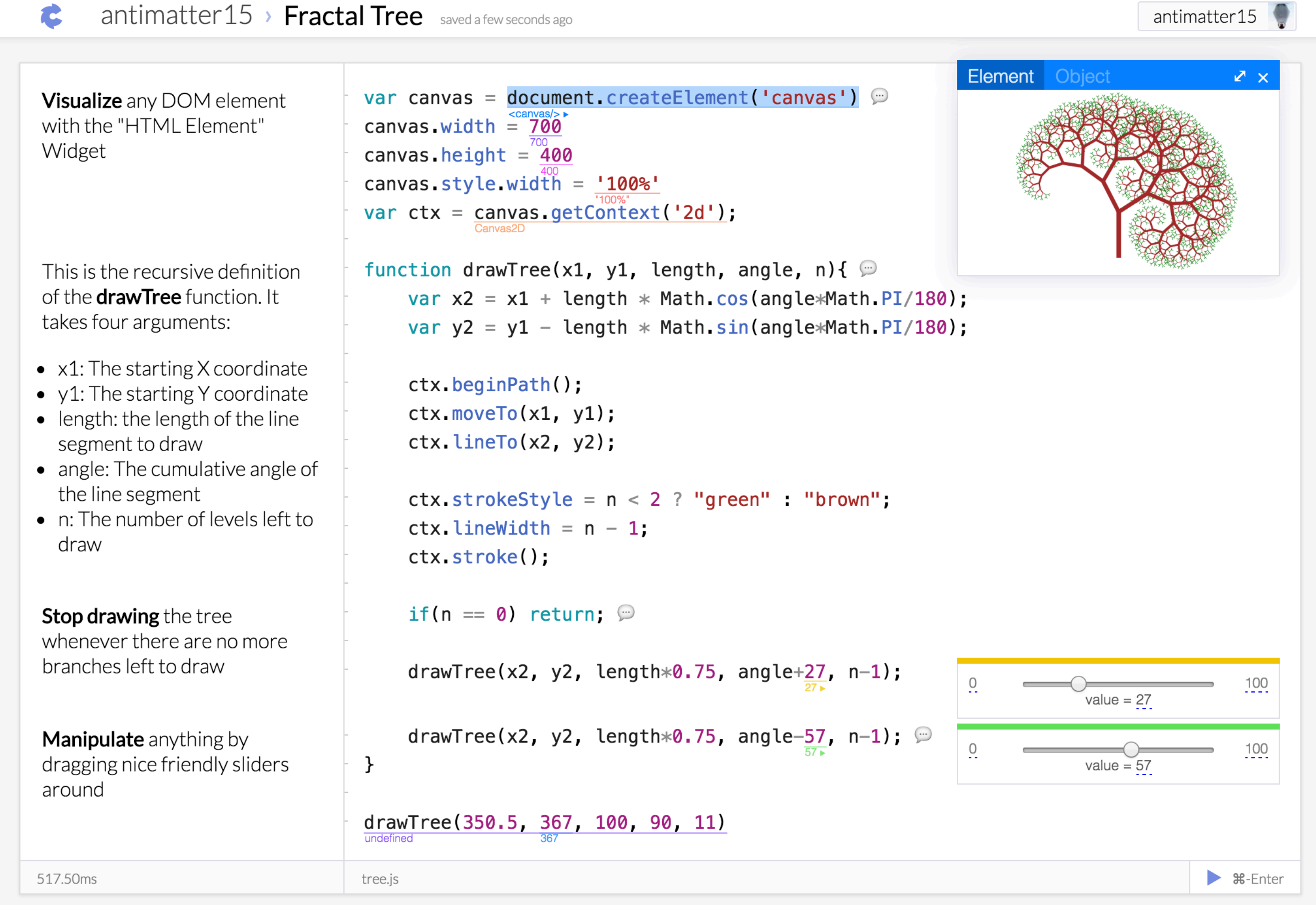
What I would push further on:
• Make data more tangible: allow me to reorder and stretch the characters.
• Make more of the algorithm tangible. e.g. A butterfly flying overhead that can also be moved; A comparator function that is somehow tangible.
• Start and build up from simplest thing: two characters to play with.
• Even simpler: play starts by manually sorting two elements themselves (no comparator and auto-jump; just click to jump and swap.)

Glen's project is unpublished. Contact him and he might send you a link.

Carbide

Guillermo Webster and Kevin Kwok (2016)

Experimental programming environment with a variety of intriguing features. (I can't get the environment to load, and the site CSS is periodically failing for me).



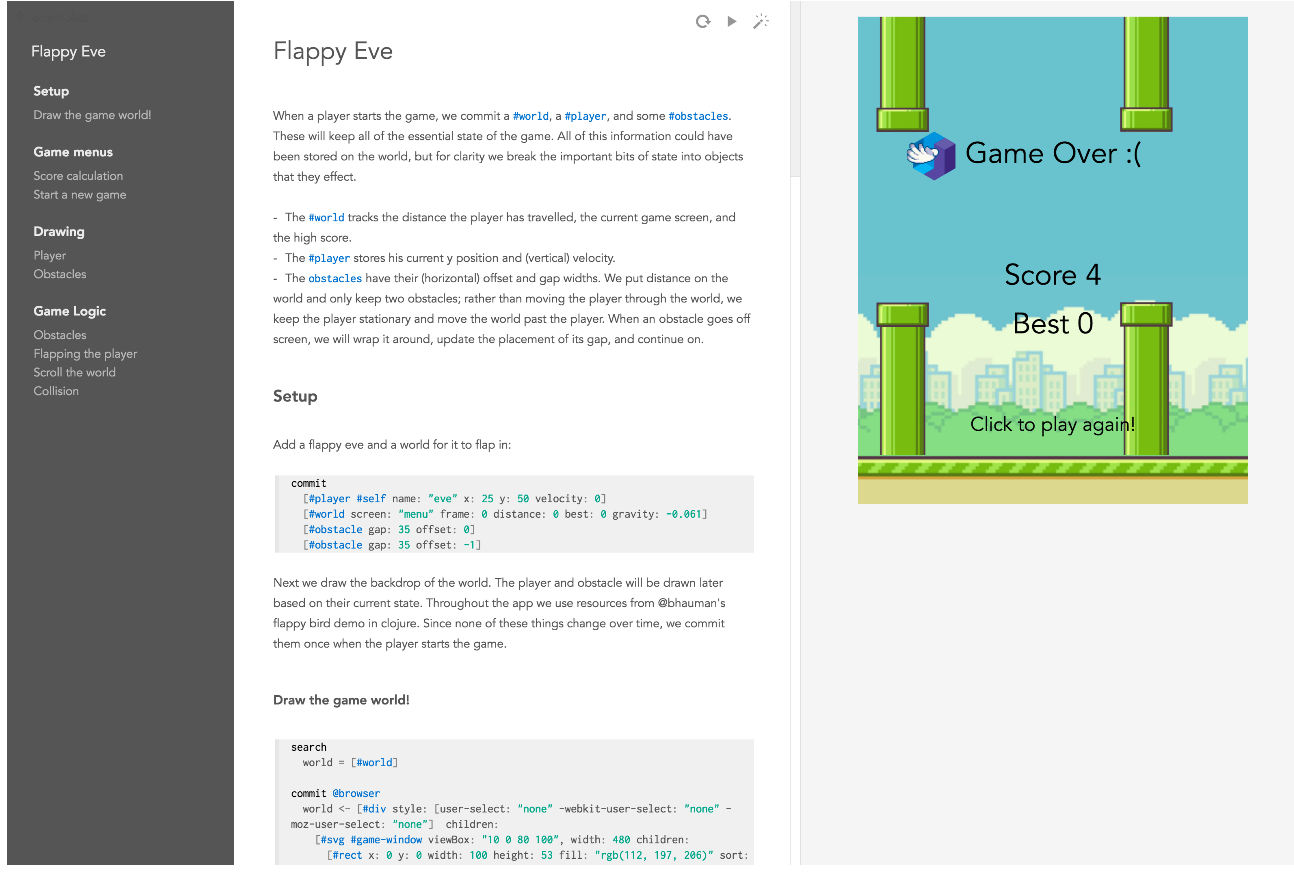
- **Back propagation** (edit program output and input changes)
- Add **probes** to the variables, expressions, or subexpressions of running programs.
- **Extensible widgets for a variety of data types** that visualize and edit these values, e.g. numbers, sliders, HTML, map data, colors, graphics, strings, matrices, JavaScript object, JSON, binary data, etc…
- Supports (in theory) any language, although JavaScript is most supported right now.
- **Programming notebook** style cells (can't find a visual of this though). They point out you need fewer cells that other environments since so much visualization is handled for you.
- Rich text comments. (Pictures would be even better).
- Polished visuals and interaction design.

http://alpha.trycarbide.com

Eve

Kodowa (2015)

Hyper-literate programming environment with a unique database/causality model. Descends from the Light Table (2012) project.



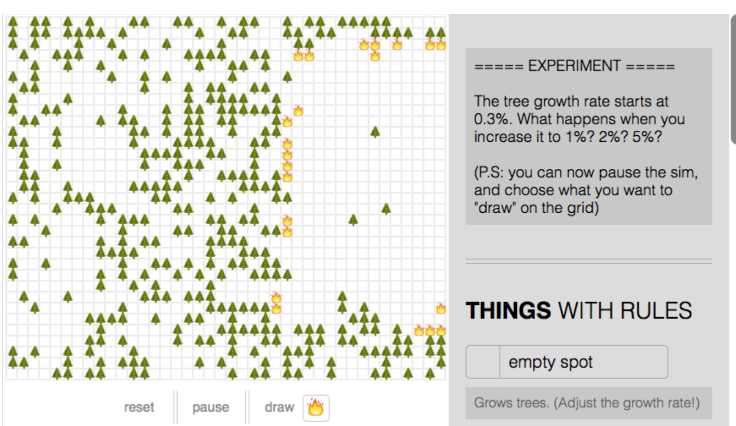
• **State database**. All state is in a database of records. Program fragments read and write to this database, which also encapsulates errors. In this fashion, complex programs are build by composing simple processes that read/write to the database. (Analogous, in some ways, to a spreadsheet.)
• **Causality tracking.** Database enables system to track causality—what led to what. (This design brings Realtalk to mind.)
• **Inspect output** (e.g. HTML) to see what might have created it.
• **Hyperliterate programming**. Code is not just embedded in prose, but a complex program takes the form of navigable hierarchical prose (i.e. a book). Code view is dynamic: select which parts of the program you want to see.
• **Inline errors**
• **Inline data.** Easy inline data visualization (notebook style), with some simple widgets, like bar graphs.

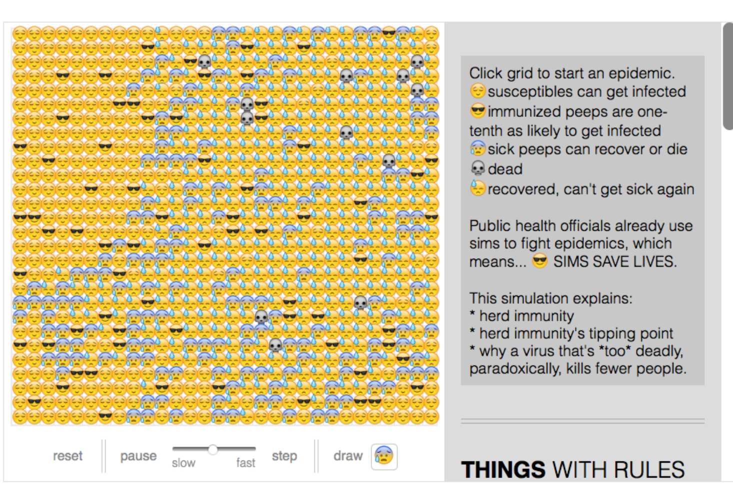
http://witheve.com

Simulating the World (in Emoji)

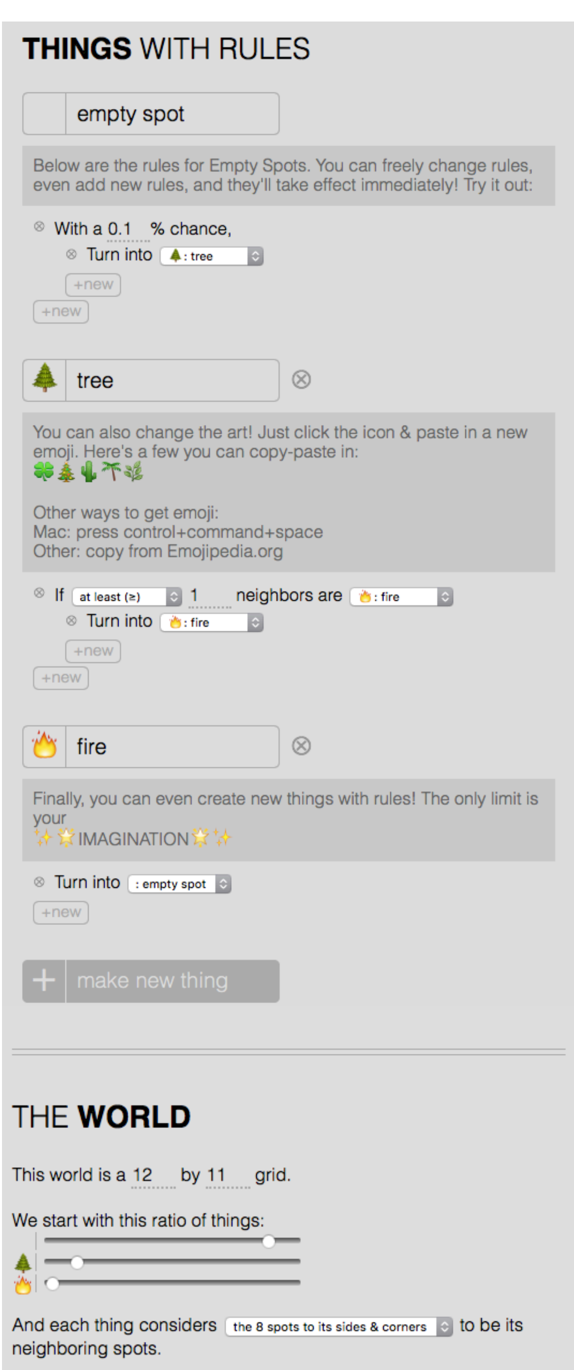
Nicky Case (2016)

An authoring tool for cellular automata style simulations.
Leverages appeal, art, and abstraction of emoji.





The world state and simulation rules can be edited live.

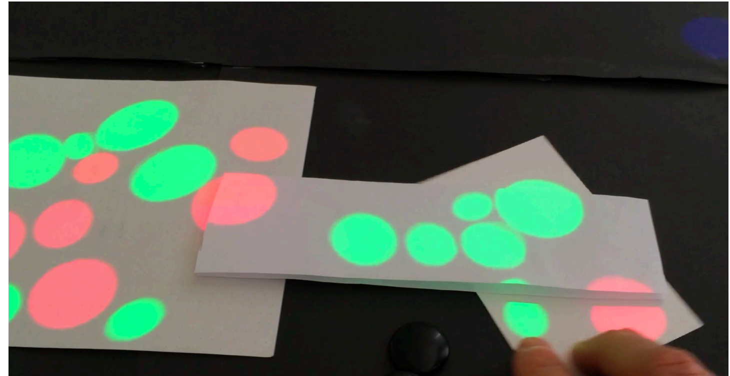


A GUI editor scaffolds creation of valid rules.

La Tabla

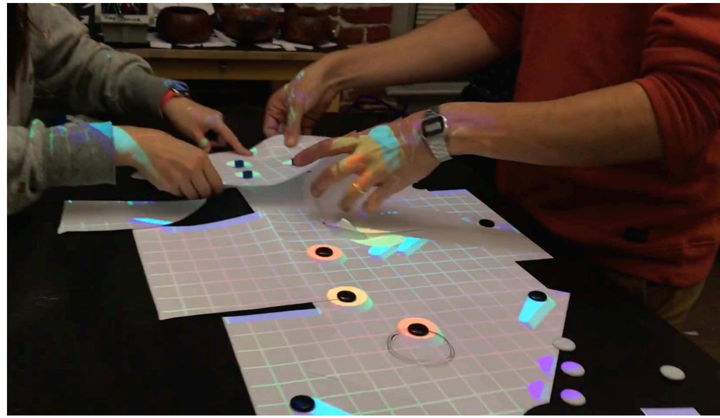
Chaim Gingold and Luke Iannini (2017)

La Tabla is an experiment in making a computerized plaything that is radically embodied and open ended. It is a magical table—put things on it and they come to life. Make music, play pong, design and play your own pinball tables, and create animations with your body, your friends, paper, drawings, game pieces—whatever strikes your fancy. La Tabla achieves this by combining computer vision, projection mapping, and design principles that anticipate and encourage open ended play and appropriation.

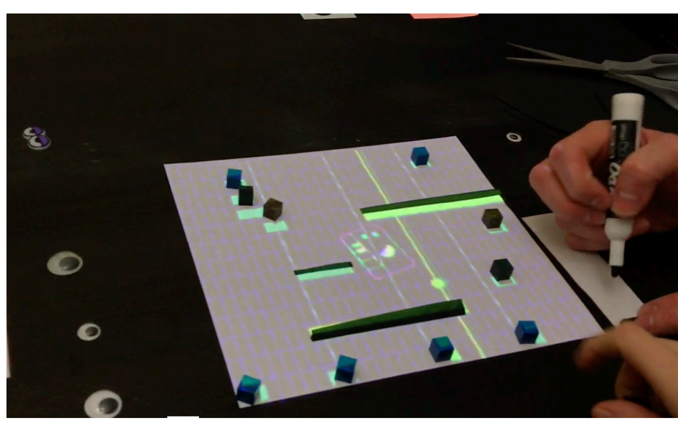


Bouncing balls.

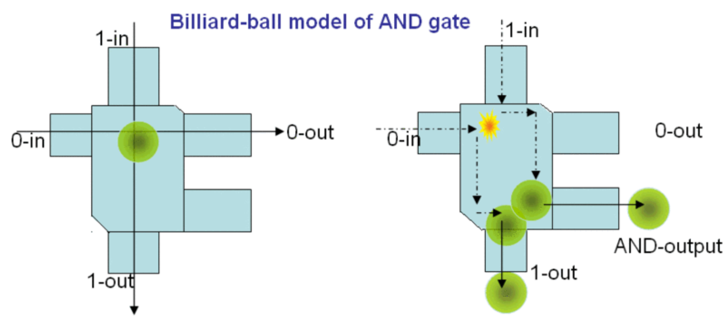
Cel animation.



Pinball construction and play.



Music scoring.

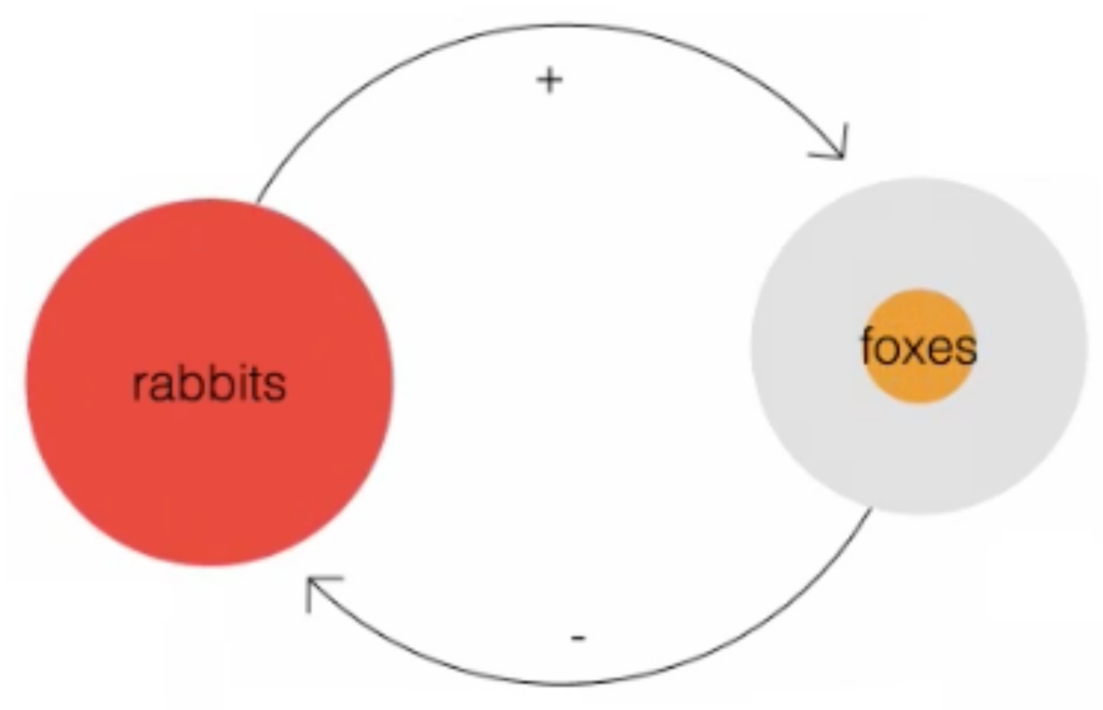


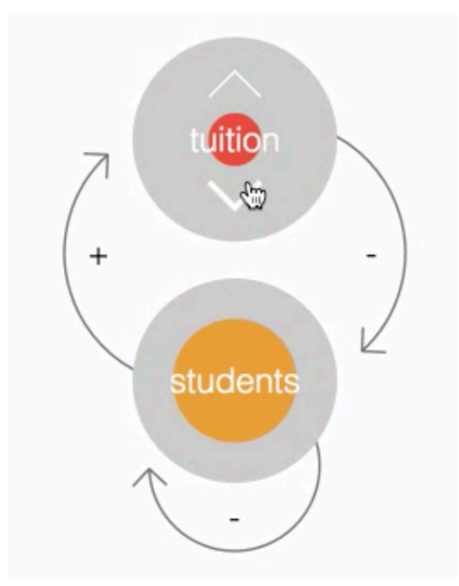
Many of Tabla's activities can be seen as a kind of programming. For example, Fredkin and Toffoli's billiard ball AND gate shows how computer logic can arise from from billiard ball physics (image from Wikipedia).

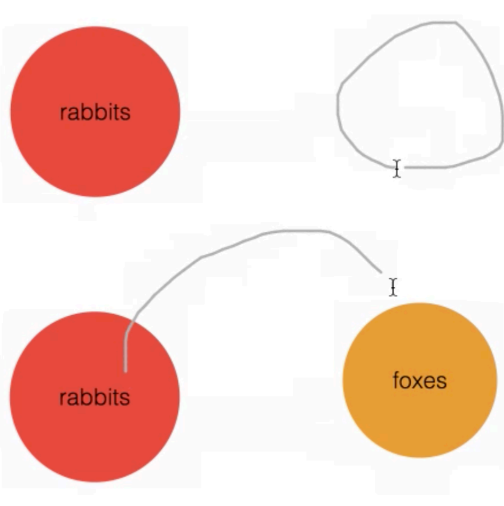
Loopy

Nicky Case (2017)

Draw circles, lines, and type to create simple system dynamic style feedback loops. In this effort, there are echoes of Richmond's Stella authoring tool (1985) and Forrester's visual system dynamics notation.







State Machine

Terry Cavanagh and Ruari O'Sullivan (2017, unreleased)

The developers of this title are experienced independent game developers. Although they've published very little about this title, it looks promising.

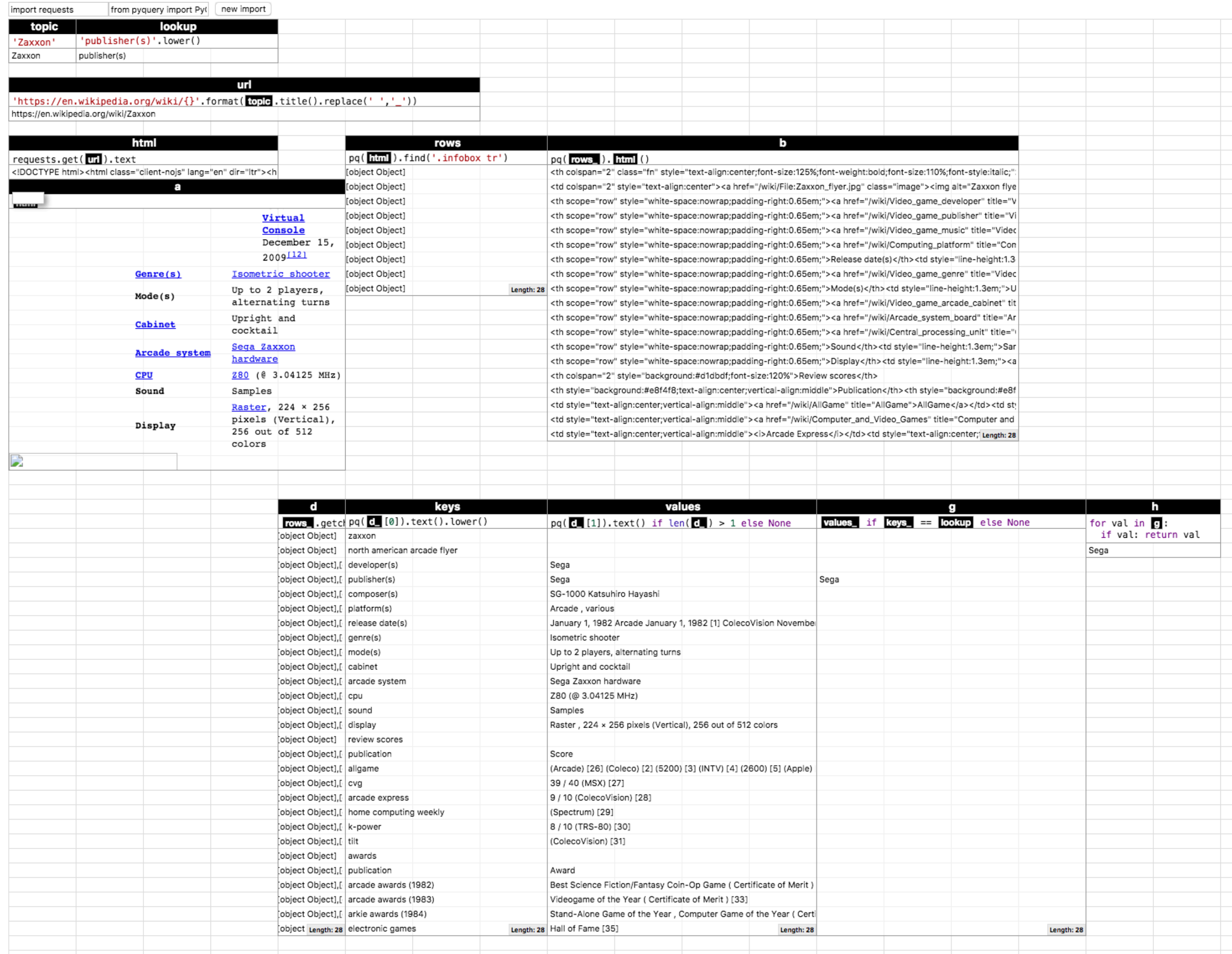
An electrical system with visually apparent state (wires on/off) helps to integrate various game systems.

http://statemachinegame.com/blog/

Flow Sheets

Glen Chiacchieri (2017)

The idea behind Flow Sheets is to make visible all of the data that courses through a program, and to afford a style of programming in which data is always visible. The resemblance to spreadsheets is more than visual: programming is never divorced from data, even while it is being authored.



Each table has a name, can be independently moved around the grid, and reference one another's data. The presentation style of tables can be changed, for example allowing html to be shown as code or a rendered page layout. When values are modified downstream data changes are called out via a simple highlighting animation.

Work in progress. This is the second major iteration of Flow Sheets. Glen created the first one in 2016.

References

Agalianos, Angelos, Geoff Whitty, and Richard Noss. “The Social Shaping of Logo.” Social Studies of Science 36, no. 2 (2006): 241–267.

Auerbach, David. “The Hardest Computer Game of All Time.” Slate, January 24, 2014. http://www.slate.com/articles/technology/bitwise/2014/01/robot_odyssey_the_hardest_computer_game_of_all_time.html.

Begel, Andrew. “LogoBlocks: A Graphical Programming Language for Interacting with the World.” Electrical Engineering and Computer Science Department, MIT, Boston, MA, 1996.

Ben-Ari, Mordechai, Roman Bednarik, Ronit Ben-Bassat Levy, Gil Ebel, Andrés Moreno, Niko Myller, and Erkki Sutinen. “A Decade of Research and Development on Program Animation: The Jeliot Experience.” Journal of Visual Languages & Computing 22, no. 5 (2011): 375–384.

Borning, Alan. “Graphically Defining New Building Blocks in ThingLab.” Human-Computer Interaction 2, no. 4 (1986): 269–295.

———. “The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory.” ACM Transactions on Programming Languages and Systems (TOPLAS) 3, no. 4 (1981): 353–387.

———. “ThingLab: A Constraint-Oriented Simulation Laboratory.” XEROX: Palo Alto Research Center, 1979.

———. “ThingLab: An Object-Oriented System for Building Simulations Using Constraints.” In Proceedings of the 5th International Joint Conference on Artificial Intelligence-Volume 1, 497–498. Morgan Kaufmann Publishers Inc., 1977. http://dl.acm.org/citation.cfm?id=1624545.

Brennan, Karen, and Mitchel Resnick. “New Frameworks for Studying and Assessing the Development of Computational Thinking.” In Proceedings of the 2012 Annual Meeting of the American Educational Research Association, Vancouver, Canada, 1–25, 2012. http://scratched.gse.harvard.edu/ct/files/AERA2012.pdf.

Brusilovsky, Peter, Eduardo Calabrese, Jozef Hvorecky, Anatoly Kouchnirenko, and Philip Miller. “Mini-Languages: A Way to Learn Programming Principles.” Education and Information Technologies 2, no. 1 (1997): 65–83.

Chalcraft, Adam, and Michael Greene. “Train Sets.” Eureka 53 (1994): 5–12.

Chambers, Craig, and David Ungar. “Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language.” In ACM SIGPLAN Notices, 24:146–160. ACM, 1989. http://dl.acm.org/citation.cfm?id=74831.

Chugh, Ravi. “Prodirect Manipulation: Bidirectional Programming for the Masses.” In Proceedings of the 38th International Conference on Software Engineering Companion, 781–784. ACM, 2016. http://dl.acm.org/citation.cfm?id=2889210.

Chugh, Ravi, Brian Hempel, Mitchell Spradlin, and Jacob Albers. “Programmatic and Direct Manipulation, Together at Last.” In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, 341–354. ACM, 2016. http://dl.acm.org/citation.cfm?id=2908103.

Cooper, Stephen, Wanda Dann, and Randy Pausch. “Alice: A 3-D Tool for Introductory Programming Concepts.” In Journal of Computing Sciences in Colleges, 15:107–116. Consortium for Computing Sciences in Colleges, 2000. http://dl.acm.org/citation.cfm?id=364161.

Cypher, Allen, and David Canfield Smith. “KidSim: End User Programming of Simulations.” In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 27–34. ACM Press/Addison-Wesley Publishing Co., 1995. http://dl.acm.org/citation.cfm?id=223908.

Drummond, Brian, and Marilyn Stelzner. "SimKit: A Model-Building Simulation Toolkit." In AI Tools and Techniques, edited by Mark H. Richer, 241, 1989. https://books.google.com/books?hl=en&lr=&id=iMUfTzVuasUC&oi=fnd&pg=PA241&dq=simkit+a+model+building+simulation+toolkit+drummond&ots=hKigENJHUh&sig=McoA2WtoTj1HJewIWgDeRuisU1c.

Du Boulay, Benedict. "Some Difficulties of Learning to Program." Journal of Educational Computing Research 2, no. 1 (1986): 57–73.

Gilmore, David J., Karen Pheasey, Jean Underwood, and Geoffrey Underwood. "Learning Graphical Programming: An Evaluation of KidSim™." In Human—Computer Interaction, 145–150. Springer, 1995. http://link.springer.com/chapter/10.1007/978-1-5041-2896-4_24.

Green, Thomas R. G., and Marian Petre. "Usability Analysis of Visual Programming Environments: A 'cognitive Dimensions' Framework." Journal of Visual Languages & Computing 7, no. 2 (1996): 131–174.

Guo, Philip J. "Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education." In Proceeding of the 44th ACM Technical Symposium on Computer Science Education, 579–584. ACM, 2013. http://dl.acm.org/citation.cfm?id=2445368.

Guzdial, Mark. "Programming Environments for Novices." Computer Science Education Research 2004 (2004): 127–154.

Guzdial, Mark, and Elliot Soloway. "Teaching the Nintendo Generation to Program." Communications of the ACM 45, no. 4 (2002): 17–21.

Hoc, J.-M. Psychology of Programming. Academic Press, 2014. https://books.google.com/books?hl=en&lr=&id=NkOjBQAAQBAJ&oi=fnd&pg=PP1&dq=Lowering+the+Barriers+to+Programming&ots=zT0A2L4u12&sig=mz5NVhAEiyaC_rhqT2cuU179p4k.

Hollan, James D., Edwin L. Hutchins, and Louis Weitzman. "STEAMER: An Interactive Inspectable Simulation-Based Training System." AI Magazine 5, no. 2 (1984): 15.

Horn, Michael S., and Robert JK Jacob. "Tangible Programming in the Classroom with Tern." In CHI'07 Extended Abstracts on Human Factors in Computing Systems, 1965–1970. ACM, 2007. http://dl.acm.org/citation.cfm?id=1240933.

Hutchins, Edwin, J. D. Hollan, and D. A. Norman. "Direct Manipulation Interfaces." Human-Computer Interaction 1, no. 4 (1985): 311–338.

Ingalls, Dan, Bert Freudenberg, Ted Kaehler Yoshiki Ohshima, and Alan Kay. "Reviving Smalltalk-78." Accessed January 17, 2017. http://esug.org/data/ESUG2014/IWST/Papers/iwst2014_Reviving%20Smalltalk-78.pdf.

Ingalls, Dan, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself." In ACM SIGPLAN Notices, 32:318–326. ACM, 1997. http://dl.acm.org/citation.cfm?id=263754.

Ingalls, Dan, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. "Fabrik: A Visual Programming Environment." In ACM SIGPLAN Notices, 23:176–190. ACM, 1988. http://dl.acm.org/citation.cfm?id=62100.

Ingalls, Daniel HH. "Design Principles behind Smalltalk." BYTE Magazine 6, no. 8 (1981): 286–298.

———. "The Smalltalk-76 Programming System Design and Implementation." In Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 9–16. ACM, 1978. http://dl.acm.org/citation.cfm?id=512762.

Ingalls, Daniel, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. "The Lively Kernel a Self-Supporting System on a Web Page." In Self-Sustaining Systems, 31–50. Springer, 2008. http://link.springer.com/chapter/10.1007/978-3-540-89275-5_2.

Jenkins, Tony. "On the Difficulty of Learning to Program." In Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences, 4:53–58. Citeseer, 2002. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.596.9994&rep=rep1&type=pdf.

Jernigan, Will, Amber Horvath, Michael Lee, Margaret Burnett, Taylor Cuilty, Sandeep Kuttal, Anicia Peters, Irwin Kwan, Faezeh Bahmani, and Andrew Ko. "A Principled Evaluation for a Principled Idea Garden." In Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on, 235–243. IEEE, 2015. http://ieeexplore.ieee.org/abstract/document/7357222/.

Kafai, Yasmin B., and Yasmin Bettina Kafai. Minds in Play: Computer Game Design as a Context for Children's Learning. Routledge, 1995. https://books.google.com/books?hl=en&lr=&id=Ocyllxa8ZjkC&oi=fnd&pg=PR2&dq=related:YCK2JCuULtAJ:scholar.google.com/

&ots=0xFjwNEXm7&sig=1-T48emVebB33aQv9EHCxtX3cbA.

Kahn, Ken. “Toontalk TM—an Animated Programming Environment for Children.” Journal of Visual Languages & Computing 7, no. 2 (1996): 197–217.

Kay, Alan. “The Early History of Smalltalk.” SIGPLAN Not. 28, no. 3 (March 1993): 69–95. doi:10.1145/155360.155364.

Kelleher, Caitlin. “Motivating Programming: Using Storytelling to Make Computer Programming Attractive to Middle School Girls.” DTIC Document, 2006. http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA492489.

Kelleher, Caitlin, and Randy Pausch. “Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers.” ACM Computing Surveys (CSUR) 37, no. 2 (2005): 83–137.

Kelleher, Caitlin, Randy Pausch, and Sara Kiesler. “Storytelling Alice Motivates Middle School Girls to Learn Computer Programming.” In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 1455–1464. ACM, 2007. http://dl.acm.org/citation.cfm?id=1240844.

Ko, Andrew J., Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, et al. “The State of the Art in End-User Software Engineering.” ACM Computing Surveys (CSUR) 43, no. 3 (2011): 21.

Ko, Andrew J., Brad A. Myers, and Htet Htet Aung. “Six Learning Barriers in End-User Programming Systems.” In Visual Languages and Human Centric Computing, 2004 IEEE Symposium on, 199–206. IEEE, 2004. http://ieeexplore.ieee.org/abstract/document/1372321/.

Krahn, Robert, Dan Ingalls, Robert Hirschfeld, Jens Lincke, and Krzysztof Palacz. “Lively Wiki a Development Environment for Creating and Sharing Active Web Content.” In Proceedings of the 5th International Symposium on Wikis and Open Collaboration, 9. ACM, 2009. http://dl.acm.org/citation.cfm?id=1641324.

Lahtinen, Essi, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. “A Study of the Difficulties of Novice Programmers.” In Acm Sigcse Bulletin, 37:14–18. ACM, 2005. http://dl.acm.org/citation.cfm?id=1067453.

Lincke, Jens, Robert Krahn, Dan Ingalls, and Robert Hirschfeld. “Lively Fabrik a Web-Based End-User Programming Environment.” In 2009 Seventh International Conference on Creating, Connecting and Collaborating through Computing, 11–19. IEEE, 2009. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5350243.

———. “Lively Fabrik a Web-Based End-User Programming Environment.” In Creating, Connecting and Collaborating through Computing, 2009. C5’09. Seventh International Conference on, 11–19. IEEE, 2009. http://ieeexplore.ieee.org/abstract/document/5350243/.

Lincke, Jens, Robert Krahn, Dan Ingalls, Marko Roder, and Robert Hirschfeld. “The Lively PartsBin–A Cloud-Based Repository for Collaborative Development of Active Web Content.” In System Science (HICSS), 2012 45th Hawaii International Conference on, 693–701. IEEE, 2012. http://ieeexplore.ieee.org/abstract/document/6148978/.

Ludolph, Frank, Y.-Y. Chow, Dan Ingalls, Scott Wallace, and Ken Doyle. “The Fabrik Programming Environment.” In Visual Languages, 1988., IEEE Workshop on, 222–230. IEEE, 1988. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=18032.

Malan, David J., and Henry H. Leitner. “Scratch for Budding Computer Scientists.” ACM SIGCSE Bulletin 39, no. 1 (2007): 223–227.

Maloney, John, Leo Burd, Yasmin Kafai, Natalie Rusk, Brian Silverman, and Mitchel Resnick. “Scratch: A Sneak Preview [Education].” In Creating, Connecting and Collaborating through Computing, 2004. Proceedings. Second International Conference on, 104–109. IEEE, 2004. http://ieeexplore.ieee.org/abstract/document/1314376/.

Maloney, John H., Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. Programming by Choice: Urban Youth Learning Programming with Scratch. Vol. 40. 1. ACM, 2008. http://dl.acm.org/citation.cfm?id=1352260.

Maloney, John H., and Randall B. Smith. “Directness and Liveness in the Morphic User Interface Construction Environment.” In Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology, 21–28. ACM, 1995. http://dl.acm.org/citation.cfm?id=215636.

Maloney, John, and Walt Disney Imagineering. “An Introduction to Morphic: The Squeak User Interface Framework.” Squeak: OpenPersonal Computing and Multimedia, 2001. http://thelackthereof.org/docs/library/unsorted/programming/morphic.final.pdf.

Maloney, John, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. “The Scratch Programming Language and Environment.” ACM Transactions on Computing Education (TOCE) 10, no. 4 (2010): 16.

Mann, Yotam, Jeff Lubow, and Adrian Freed. “The Tactus: A Tangible, Rhythmic Grid Interface Using Found-Objects.” In NIME, 86–89, 2009. http://www.academia.edu/download/43631493/nm090075.pdf.

In NIME, 86–89, 2007. http://www.academia.edu/download/[illegible].pdf.
Martin, F., G. L. Colobong, and M. Resnick. Tangible Programming with Trains, 1999.
McNerney, Timothy S. “From Turtles to Tangible Programming Bricks: Explorations in Physical Language Design.” Personal and Ubiquitous Computing 8, no. 5 (2004): 326–337.
———. “Tangible Programming Bricks: An Approach to Making Programming Accessible to Everyone.” Thesis, Massachusetts Institute of Technology, 2000. http://dspace.mit.edu/handle/1721.1/62094.
Meerbaum-Salant, Orni, Michal Armoni, and Mordechai Ben-Ari. “Learning Computer Science Concepts with Scratch.” Computer Science Education 23, no. 3 (2013): 239–264.
Moloney, J., Alan Borning, and Bjorn Freeman-Benson. Constraint Technology for User-Interface Construction in ThingLab II. Vol. 24. 10. ACM, 1989. http://dl.acm.org/citation.cfm?id=74917.
Morgado, Leonel, Maria Cruz, and Ken Kahn. “Radia Perlman—A Pioneer of Young Children Computer Programming.” Current Developments in Technology-Assisted Education. Proceedings of M-ICTE, 2006, 1903–1908.
Moskal, Barbara, Deborah Lurie, and Stephen Cooper. “Evaluating the Effectiveness of a New Instructional Approach.” ACM SIGCSE Bulletin 36, no. 1 (2004): 75–79.
Nardi, Bonnie A. A Small Matter of Programming: Perspectives on End User Computing. MIT press, 1993. https://books.google.com/books?hl=en&lr=&id=0drDRT370eoC&oi=fnd&pg=PR11&dq=nardi+spreadsheet&ots=eFnU_hPwnu&sig=rdsz4l9pZT7ClYVcAm3mMJiY56g.
Nardi, Bonnie A., and James R. Miller. “An Ethnographic Study of Distributed Problem Solving in Spreadsheet Development.” In Proceedings of the 1990 ACM Conference on Computer-Supported Cooperative Work, 197–208. ACM, 1990. http://dl.acm.org/citation.cfm?id=99355.
———. The Spreadsheet Interface: A Basis for End User Programming. Hewlett-Packard Laboratories, 1990. http://www.miramontes.com/writing/spreadsheet-eup/.
———. “Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development.” International Journal of Man-Machine Studies 34, no. 2 (1991): 161–184.
Nickerson, Jeffrey. “Visual Programming,” 1994.
Pane, John, and Brad Myers. “Usability Issues in the Design of Novice Programming Systems,” 1996. http://repository.cmu.edu/isr/820/?utm_source=repository.cmu.edu%2Fisr%2F820&utm_medium=PDF&utm_campaign=PDFCoverPages.
Papert, Seymour. Mindstorms: Children, Computers, and Powerful Ideas. New York: Basic Books, 1980.
Pattis, Richard E. Karel the Robot: A Gentle Introduction to the Art of Programming. John Wiley & Sons, Inc., 1981. http://dl.acm.org/citation.cfm?id=539521.
Pears, Arnold, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. “A Survey of Literature on the Teaching of Introductory Programming.” ACM SIGCSE Bulletin 39, no. 4 (2007): 204–223.
Perlman, Radia. “Using Computer Technology to Provide a Creative Learning Environment for Preschool Children (PDF Download Available).” ResearchGate. Accessed January 31, 2017. https://www.researchgate.net/publication/37596649_Using_Computer_Technology_to_Provide_a_Creative_Learning_Environment_for_Preschool_Children.
Rajala, Teemu, Mikko-Jussi Laakso, Erkki Kaila, and Tapio Salakoski. “VILLE: A Language-Independent Program Visualization Tool.” In Proceedings of the Seventh Baltic Sea Conference on Computing Education Research-Volume 88, 151–159. Australian Computer Society, Inc., 2007. http://dl.acm.org/citation.cfm?id=2449340.
Repenning, Alex. “Agentsheets,” 1993.
———. “Agentsheets: A Tool for Building Domain-Oriented Visual Programming Environments.” In Proceedings of the INTERACT’93 and CHI’93 Conference on Human Factors in Computing Systems, 142–143. ACM, 1993. http://dl.acm.org/citation.cfm?id=169119.
Repenning, Alexander. “AgentSheets®: An Interactive Simulation Environment with End-User Programmable Agents.” Interaction, 2000. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.36.2039&rep=rep1&type=pdf.
Repenning, Alexander, Andri Ioannidou, and John Zola. “AgentSheets: End-User Programmable Simulations.” Journal of Artificial Societies and Social Simulation 3, no. 3 (2000): 351–358.
Repenning, Alexander, and Tamara Sumner. “Agentsheets: A Medium for Creating Domain-Oriented Visual Languages.” Computer 28, no. 3 (1995): 17–25.
Repenning, Alexander, David Webb, and Andri Ioannidou. “Scalable Game Design and the Development of a Checklist for Getting Computational Thinking into Public Schools.” In Proceedings of the 41st ACM Technical

Symposium on Computer Science Education, 265–269. ACM, 2010. http://dl.acm.org/citation.cfm?id=1734357.

Resnick, Mitchel. "StarLogo: An Environment for Decentralized Modeling and Decentralized Thinking." In Conference Companion on Human Factors in Computing Systems, 11–12. ACM, 1996. http://dl.acm.org/citation.cfm?id=257095.

Resnick, Mitchel, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, et al. "Scratch: Programming for All." Communications of the ACM 52, no. 11 (2009): 60–67.

Resnick, Mitchel, and Brian Silverman. "Some Reflections on Designing Construction Kits for Kids." In Proceedings of the 2005 Conference on Interaction Design and Children, 117–122. ACM, 2005. http://dl.acm.org/citation.cfm?id=1109556.

Richmond, Barry. "STELLA: Software for Bringing System Dynamics to the Other 98%." In Proceedings of the 1985 International Conference of the System Dynamics Society: 1985 International System Dynamics Conference, 706–718, 1985.

Robins, Anthony, Janet Rountree, and Nathan Rountree. "Learning and Teaching Programming: A Review and Discussion." Computer Science Education 13, no. 2 (2003): 137–172.

Sloman, Aaron. "Interactions between Philosophy and Artificial Intelligence: The Role of Intuition and Non-Logical Reasoning in Intelligence." Artificial Intelligence 2, no. 3–4 (1971): 209–225.

Smith, David Canfield. "Pygmalion: A Creative Programming Environment." Stanford University, 1975.

Smith, David Canfield, Allen Cypher, and Jim Spohrer. "KidSim: Programming Agents without a Programming Language." Communications of the ACM 37, no. 7 (1994): 54–67.

Soloway, Elliot. "Learning to Program= Learning to Construct Mechanisms and Explanations." Communications of the ACM 29, no. 9 (1986): 850–858.

Soloway, Elliot, and James C. Spohrer. Studying the Novice Programmer. Psychology Press, 2013. https://books.google.com/books?hl=en&lr=&id=vFhEAgAAQBAJ&oi=fnd&pg=PP1&dq=related:YCK2JCuULtAJ:scholar.google.com/&ots=W21hSJnEA0&sig=gw8EMZQYc1bCcvqOc27j_5DMJU0.

Sorva, Juha. "Notional Machines and Introductory Programming Education." ACM Transactions on Computing Education (TOCE) 13, no. 2 (2013): 8.

———. Visual Program Simulation in Introductory Programming Education. Aalto University, 2012. https://aaltodoc.aalto.fi/handle/123456789/3534.

Sorva, Juha, Ville Karavirta, and Lauri Malmi. "A Review of Generic Program Visualization Systems for Introductory Programming Education." ACM Transactions on Computing Education (TOCE) 13, no. 4 (2013): 15.

Sorva, Juha, Jan Lönnberg, and Lauri Malmi. "Students' Ways of Experiencing Visual Program Simulation." Computer Science Education 23, no. 3 (2013): 207–238.

Sorva, Juha, and Teemu Sirkiä. "UUhistle: A Software Tool for Visual Program Simulation." In Proceedings of the 10th Koli Calling International Conference on Computing Education Research, 49–54. ACM, 2010. http://dl.acm.org/citation.cfm?id=1930471.

Stewart, Ian. "Commuters and Computers: The Intelligent Subway," 2008.

Suzuki, Hideyuki, and Hiroshi Kato. "AlgoBlock: A Tangible Programming Language, a Tool for Collaborative Learning." In Proceedings of 4th European Logo Conference, 297–303, 1993. https://www.researchgate.net/profile/Hideyuki_Suzuki5/publication/242383829_Algoblock_a_tangible_programming_language_a_tool_for_collaborative_learning/links/575f5c8e08ae414b8e5496e3.pdf.

Weinberg, Gerald M. The Psychology of Computer Programming: Silver Anniversary Edition. Anl Sub edition. New York: Dorset House, 1998.

Williams, Michael D., James D. Hollan, and Albert L. Stevens. "Human Reasoning about a Simple Physical System." In Mental Models, edited by D. Gentner and A. Stevens, 131–154, 1983. https://books.google.com/books?hl=en&lr=&id=G8iYAgAAQBAJ&oi=fnd&pg=PA131&dq=Human+reasoning+about+a+simple+physical+system.+&ots=aLvRSVGyfA&sig=D0lN5oGnXtymcFXy4wFd-Yy0LN4.

Wing, Jeannette M. "Computational Thinking." Communications of the ACM 49, no. 3 (2006): 33–35.

———. "Computational Thinking and Thinking about Computing." Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 366, no. 1881 (2008): 3717–3725.

Winslow, Leon E. "Programming Pedagogy—a Psychological Overview." ACM Sigcse Bulletin 28, no. 3 (1996): 17–22.