Sample NSF Proposal: Computer Science
Project Summary & Description

Steps Toward The Reinvention of Programming

A Compact And Practical Model of Personal Computing As A Self-Exploratorium

Alan Kay, Dan Ingalls, Yoshiki Ohshima, Ian Piumarta, Andreas Raab

We *make*, not just to have, but to *know*

Introduction: Over the years, our research has combined interests and inventions in computer science (object-oriented programming, reflective whole systems, networking), graphics (bit-map screens and "bitblt"), UI (overlapping windows and icons, modeless editing and interactions), education (particularly for children), design (aiming for beauty as well as problem solving), and computer engineering (learning how to efficiently make whole HW & SW systems). The latter has allowed us to tackle new approaches on a large scale, deploy and test real solutions without huge teams or being bound to particular computational models, tools or platforms. Some of the alternative approaches we've invented have become mainstream (and some not), but a very important process in our work over the years is to be able to control our own HW & SW destinies by making high-enough-level languages with powerful enough tools to allow our small but critical mass research group to successfully implement anything we want to try.

In short, we've been very interested in the *ideas and ideals* of personal computing since its inception, and have strived to come up with real advances for real users, by making real systems, deploying and testing them.

These alternative systems have been small in comparison with standard practice, e.g. today's Squeak Smalltalk[Sq1] covers much of personal computing, includes its own applications, operating environment, UI and development tools, runable specifications, etc., in about 2.8MB of code (about 200,000 lines). But our intuitive sense of "mathematical entropy" insists that an even more comprehensive design approach to whole-system personal computing could be smaller by a factor of 10 or more (a factor of 2 from removing non-used code, and another factor of 5 or more via a different, more advanced architecture and design).

Possibility And Proposal: This opens the exciting possibility of *creating a practical working system that is also its own model* – a whole system *from the end-users to the metal* that could be extremely compact (we think under 20,000 lines of code) yet practical enough to serve both as a highly useful end-user system and a "system to learn about systems". I.e. the system could be compact, comprehensive, clear, high-level, and understandable enough to be an "*Exploratorium of itself*". It would:

Contribute a better approach to personal computing overall, and many individual parts of it
Provide important advances in computer science, software engineering and understanding of design
Boost pedagogical and general understanding and learning about systems and how to explain them
Create a test-bed environment for experiments into "the programming of the future" for end-users and pros

Intellectual Merits: A large number of the most important ideas and intellectual contributions of this system are new (or are non-mainstream ideas from the past that will appear new) disposed in powerful and often novel ways. Those we find particularly interesting are: how the bootstrapping is done, a "universal object" approach to end-user facilities, alternatives to "OS", "apps", "web", etc., use of roles instead of inheritance, symmetric messaging and events, invertible processes, labeled histories, distributed objects, protection, separation of meanings from optimizations, coherence and persistence of meanings, instrumentation for many kinds of self-disclosure and explanation, new ways to program for end-users and adepts, etc.

Broader Impacts: Most early learning of programming is done in a non-scalable way, somewhat equivalent to banging together a doghouse with nails and planks. What is learned doesn't scale well by a factor of 5, let alone factors of 100+. Even more critical is that the deeper mathematics-like nature of the most powerful ideas in computing are poorly described and learned via papers: even if they are read, this is a difficult form for learning and understanding. We think that making a well designed system that is also instrumented to be learned, understood and changed could have a large positive impact on many areas of computing. It would constitute an example, and a kind of "the system is the curriculum" for learning many important powerful ideas, especially for teenagers, university students, and in the 3rd world, where there is a pool of literally billions of potential computer users and authors, who could make great use of a simpler stronger approach to personal computing and explanation.

Outline of the Proposal:

1. We start by giving an abstract of what we mean by "a whole personal computing system".
2. Then we show suggestive examples of some of the design approaches we expect to take.
3. Some of the technical details and powerful principles will be discussed
4. Then we outline how we plan to go about making the system over a period of several years
5. We end with discussions of collegial relationships, education, dissemination, etc.

1. What Do We Mean By "End-User to the Metal" Personal Computing?

Compact models of fundamental ideas have aided thinking in many sciences. We think they haven't been done often enough in computing, but when done, have had similar positive effects [Mo]. We want our model to be explanatory, but we also want it to run well enough to serve as a practical artifact with a wide range of application.

A good *model* of personal computing should certainly cover today's good ideas and scales, but because it is an extreme abstraction, it does not have to reverse engineer shortfalls in unity and design; instead, it can be unified as much as possible if it covers the worthwhile parts of the end-user experiences.

We can start with our computational environment as a world-wide "every person" communications network with suitable physical input and output effectors for all users. We will address typical hardware of today, including the tantalizing "$100 laptop" that uses underpowered and scaled down resources in order to be more affordable in more parts of the world. Our "from the metal to the end-user" design and implementation will use an architectural approach that is rather different from the standard stacked layers of monolithic code from "operating system" to "applications" to "user interface". We still have to deal with issues of resource allocation and creation, internet communication, protection, graphics, sound, etc., user interface, end-user experience and functionality, etc., but we will use quite a different approach to design and underlying architecture.

"Operating Systems": We want to be able to do what operating systems do (and more), but we think there are better paths than to make a traditional classical operating system as layers of code.

Internet inspiration: Neither the computers on the Internet, nor their operating systems, nor the objects in their software systems have to be identical in order for the system to work. Instead, the heterogeneous mixture that the system is made from must simply obey message passing conventions in order to interoperate. We can note in passing that one of the biggest problems in the development of object-oriented SW architectures, particularly in the last 25 years, has been an enormous over-focus on objects and an under-focus on *messaging* (most so-called object-oriented languages don't really use the looser coupling of messaging, but instead use the much tighter "gear meshing" of procedure calls – this hurts scalability and interoperability).

Once we have truly committed to "real objects", many things are possible. For example, we can safely deal with real objects that are written in other forms via a combination of message-passing and address space confinement (we will say more about this later on). In particular, we can modularize one of the most pernicious parts of making a new operating system – namely drivers – by noting that device drivers should never have to be part of any well designed system. This is because a much better way to organize drivers is to have the devices themselves be able to furnish them to a system that needs them in a form that can be run and used safely. Again, we see that the key here is to have both the device and its drivers be logical objects on the network (and hence already able to use the universal messaging conventions that are our sole standard). An imported driver can then be run in a separate address space for safety and communicated with via standard messages.

So we will need to have some of the objects of the system be able to deal with various kinds of low-level hardware resources and help make a nicer environment for other objects and their needs. We have done quite a bit of this kind of system building over the years and explain the gist how this will be done in the section on bootstrapping ahead.

"Applications": We want to be able to do what applications do (and more), but, as with "operating systems", we think there are better paths than the traditional annoying stovepipes that give rise to a few proprietary objects in a way that makes it difficult to combine. Something more like a desktop publishing system that could allow any and all objects to be freely combined visually and behaviorly would be much better. Just as a DTP system allows many different visual elements to be formatted in a wide variety of ways (and master templates made to capture the most useful forms) to cover the entire space of user documents, we would like to go farther in this direction to cover all of the end-user's needs with a single notion of objects, graphics, user interface, publishing and search. One metaphor that might help (and was an inspiration for many of these ideas) is "HyperCard on Steroids". To do this one would extend HyperCard to have any number of useful objects, allow all to be scripted, and allow the hyperCards to be both full-fledged media pages for docs, web, and presentations, etc., **and** to recursively be its own embedded media objects. We have made systems using this approach over the last 10 years and will show a few examples further on.

Graphics, Sound, etc.: The system will provide its own low-level media models to be carried out on the hardware of the host machines. The current Squeak Smalltalk supplies a variety of graphics facilities, including a modern version of the original historical "bitblt" with rotation, alpha-blending, multi-depth color, etc., and vectorized 2D and 3D objects, roughly equivalent to Flash and OpenGL. One of the more interesting included cross platform higher-level models is an MPEG player. In our proposed new system, all of these will be consolidated into a much

simpler scheme that provides a little more functionality with much less code (some of which is described further on).

Printing: We finesse the problem of printer drivers above, but we can't finesse the problem of being able to print from our system via drivers we have obtained from the printers. We think it is fair if this is solved by generating postscript document format from our media.

The Internet And "The Web": We will certainly provide connection to the Internet, but, because the web architecture is so *ad hoc*, we will instead provide an equivalent simpler functionality that can work within the existing web. For example, a really great web-like experience would be to be able to find, view, and WYSIWYG author arbitrary multi-media "pages" that are hyperlinked, searchable and allow "services". This is really easy, and the web could have been easily designed this way, given that HyperCard was already around and thriving to provide a suggestive model when the first web browsers were made. We will follow that ignored path rather than the existing one.

Protection and Safety[Prot]**:** An important point about "real objects" is that they are already protected (they will only receive messages that they want to receive, and they will only act on messages they want to act on). The early notion of capabilities as an unforgeable filter on privileges for particular objects fits very well into a real object scheme. The capability approach to modern protection via message sending which give rise to *promises for future transactions* is very compatible with our work and with David Reed's scheme [Re] for distributed transactional protected objects that has been extensively used in the Croquet [Cr] system over the last several years (more ahead).

End-User Experiences: For the end-user, protection and safety are intimately bound up with being able to recover from any and all errors. In this system we want to go very far in dealing with "invertible processes", in part because the self-disclosing nature of the design will encourage the end-users to both look at all levels of the system and to try things, many of which would have catastrophic consequences without good facilities for real-time checkpointing, versioning, undo and redo.

User Interface Environment: interaction, widgets, etc. In addition, we think that a large part of programming language design – not just for novices, but for all who program – is treating the language and how it is worked with as *user interface design*. We address this further in the next two sections.

Computations And Programming: we wish to have as simple as possible scripting "all the way down" and we want to be able to allow multiple styles of programming to work under one logically consistent framework. We also want to experiment with and invent new ways for both end-users and professionals to program. Examples of traditional novice and expert end-user programming are spreadsheeting, HyperCard-like scripting of media, searching, making templates for new media types, etc. We will combine these into a single metaphor.

Non-traditional end-user programming includes various kinds of problem solving via conditions and constraints, massively parallel "particles and fields" techniques, "-ductions" (de-, in-, ab-, etc.), etc. Some of these will be in the system as mainstream ideas, e.g. we have had enough experience with "particles and fields" and how this style can extend from prototype programming to massively parallel loosely coupled, to have many parts of the system programmed this way. Less tested new ways to program will be exhibited as alternative curiosities. One of the most interesting (we think) alternative programming methods is how the system itself is bootstrapped (more about this ahead), and allows down to the metal changes and additions to be made as new needs arise.

Some of our best results have come from adding novices, including children, into the mix of users that need to be served. This anticipates one of the 21st century destinies for personal computing: a real computer literacy that is analogous to the reading and writing fluencies of print literacy, where all users will be able to understand and make ideas from dynamic computer representations. This will require a new approach to programming.

Explanation and Self-Disclosure: All the personal computing systems we are aware of, including our own, are very poor at explaining themselves, even to professionals, and are essentially opaque to non-expert users. A number of AI and expert systems have employed filtered versions of deductive histories to produce a narrative explanation of their inferences [Exp]. We have experimented with these ideas to allow any structure to explain and show how it was made. These simple explanations can be annotated to produce richer and more forgiving renditions, but the real value in explanations are those that can be generated automatically in a setting that allows end-user exploration. Our particular approach to modeling time (see ahead) also allows extensive undo, redo, checkpointing, and the ability to run computations both forwards and backwards to understand what is going on. Our aim here is to have "the system be the curriculum". This will eventually require this system to go beyond being reflective to being *introspective* via a self-ontology. This can be done gradually without interfering with the rest of the implementation.

2. Design Approaches

Pluralitas non est ponenda sine necessitate - William of Occam
Things should be as simple as possible, but not simpler - Albert Einstein

An important process of Design is finding the fertile ground between Occam and Einstein. We can make the physical universe from a few elementary particles and fields, but the myriad combinations of these at every level seem to give rise to quite a bit of complexity and a large number of categories[De]. Similarly we can make "computing" from a few simple relations, but this doesn't prevent systems bloat. In Biology [Bio], one of our favorite sources of fruitful analogies, the scaling of entities is not smooth but jumps from rather small carbon based molecules to much larger entire cells that can play many kinds of roles derived from very similar architectures. Looking ahead to even more interesting possible analogies with Biology are the recent advances in understanding developmental processes of multicelled animals. Quite contrary to the assumptions of biologists in the 60s – when it was thought that structures, such as eyes and legs in insects and vertebrates were likely results of parallel evolution – it has now been discovered that the basic "tool-kit" genetic apparatus for building "bodies that work" is shared by all animals (e.g. the gene that promotes the creation of an eye in a mouse will promote the creation of a (fruit fly) eye in a fruit fly.

These large plateaus for stable structures suggest we take a similar and somewhat "theatrical view" of a system in which every entity at every level is portrayed by an intelligent actor wearing appropriate costumes and simply playing a role. Here we are "not multiplying entities unnecessarily", but are putting the burden on a single kind of object (and we hope that *it* can be explained simply enough to make the larger system much easier to understand than if it had been built from many thousands of seemingly different entities. The idea here would be to jump from primitives (derived from the HW) as directly as possible to "comprehensively capable objects" and to differentiate these in ways analogous to the 250+ derived cell types in our bodies, which are all variations of the original fertilized cell.

And, since much of "personal computing" is necessarily about the experiences of "everyone as an end-user" it is important at the user interface level to create the illusion (if not the reality) of an extremely simple but wide ranging environment for making and dealing with all manner of manifested ideas. In fact, we can go farther and say the end-users are the only important faction to serve, and the value of a personal computing system comes solely from how well its users are served.

This implies that our "actors" must be comprehensible by the end-users, and the user interface and user experience is a good starting place to invent the single kind of actor.

Many of the useful entities in the user experience can be viewed in more than one way (for example, a spreadsheet cell could show a number or a visual bar; an article in a browser could have a simpler "suitable for printing" view, etc.). Are these "views" separate kinds of entities, or could they be the very same kind of actors? That is, are costumes different from humans (they certainly are in virtually all theatrical experiences) or could a "costume" also be portrayed by an actor?

In this project, beauty is especially important since we intend to allow every part of it to be examined, explained, de- and reconstructed. Combining our need for beauty with our other goal of "extremely small size without giving up power" reminds us immediately of great ideas from the past, such as. McCarthy's "Lisp in Lisp", Sutherland's separation in Sketchpad of "constraints as measures" from its solvers, Engelbart's comprehensive approach to "augmenting human intellect", Irons' insights about compiling and bootstrapping extensible languages, Strachey and Landin's approach to user friendly forms for functional programming, Atkinson's end-user UI model in HyperCard, etc. These are a few examples drawn from a *rich context of prior work* by others which include about a dozen powerful principles that we'll use to design and build this system (see **Technical Notes**, page 9, for references).

"The Many" From "The One": At the first level of authoring we want to just pick useful objects from a "supplies bin" and organize them "WYSIWYG" as we see fit to make our best statement about our topic. Hypercard has been a main influence here [HC]. We will have printing quality text in a myriad of fonts, pictures, videos, hyperlinks of various kinds, constructions of simulation models, etc. We want to make these ideas public with a single button push, as "email", as a "blog", as a "web page", etc. We want many ancillary kinds of communication to be possible, including "pen-pals", mentoring via screen-sharing, chatting, return emails, annotations and comments, etc.

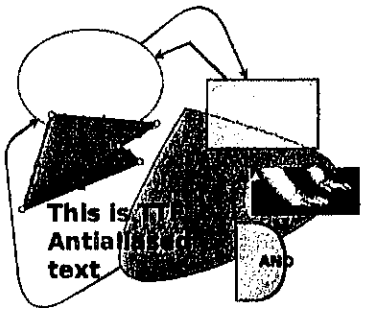
Both this first level of authoring and the next level (how all these useful objects are made) can benefit by an extreme unity in the underlying design. For example, we certainly want them to have the same behaviors as far as laying out our document is concerned. But, as we will see, they can also (a little more covertly) be the very same kind of object underneath, rather like actors on the stage wearing different costumes (including the scenery!) but all portrayed by human beings. Here are a number of examples done in our existing children's authoring system Squeak Etoys [Et].

All Objects Are "The Same" and Recursively Embeddable

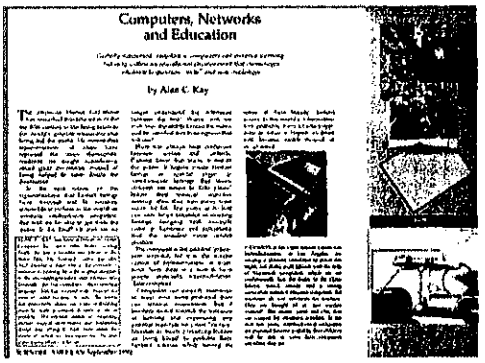
At the end-user level, all objects seem to be made from a single kind of object – visibly, a kind of smart polygon that can have holes, be smoothed and filled, recursively embedded and scripted, etc. Like the theater in Shakespeare's time, in which even the trees and other scenery were people wearing costumes (pretty handy for changing scenes), it is possible to have all the objects be "the same" where the slight differences are in look and specific role, and more parametric than in specie.

"No Apps" – Instead, Any Of The Existing and Created Objects Can Be Arbitrarily Combined

Separate applications are an old 60s idea which force a "stovepipe" mode of authoring that is limited to what the application can do, and this makes new ideas by end-users difficult to fit in. Looking at it from this point of view, there is only one "application" in our model – itself – and all old and new things are created, manipulated and presented there.

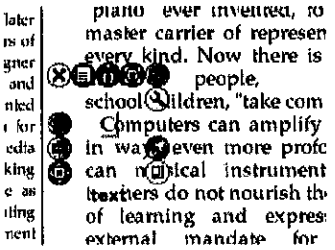


The many graphical costumes are all made from the same universal shape, and include ovals, rectangles, polygons and curves, pictures and drawings, TTF antialiased text that can flow from one container to another (for DTP), and smart connectors for making diagrams. Any object can hold any other

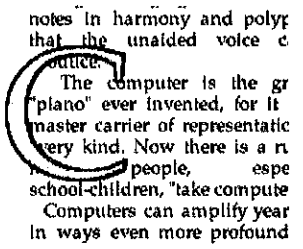


"Desktop Publishing" is not an "application" but simply an organization of desired objects to look like a high-quality printed page. Any old or new objects can be simply dragged to become part of this "document".

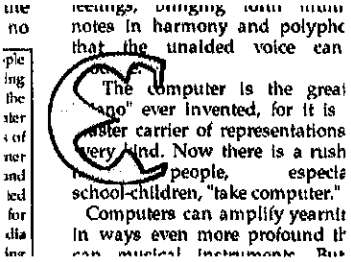
All the objects are "the same object". Here we click on a frame that contains text flow.



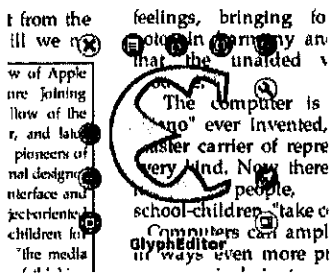
We get the same halo on the capital "C"



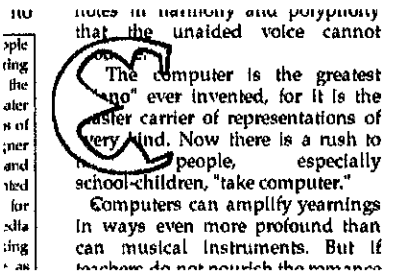
Diving into the character gives us the smoothed polygon that is the basis of all 2D graphics (and TTF Fonts).



This is itself editable



... and it is a standard Etoy object, but playing a different role.



We can apply the change locally to our selected capital C...

Computers, Networks and Education

Globally networked, easy-to-use computers can enhance learning, but only within an educational environment that encourages students to question "facts" and seek challenges

by Alan C. Kay

... or globally to all the capital C's in Palatino Linotype

All Objects Are Scripted "All The Way Down"

Our model system will use a deeper refinement of the children's environment, but it will have enough of a similar flavor to motivate showing a few examples here. First let's paint a simple object, explore it, and then think about what things would be like if all of the objects in the system were the same.

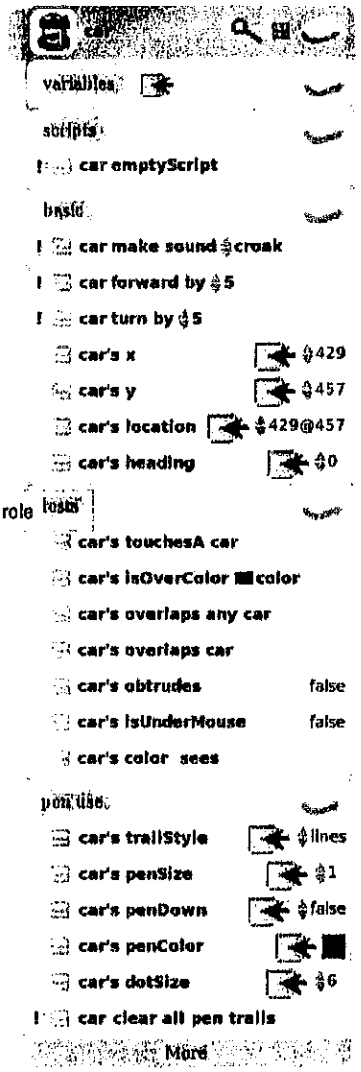
The Handle Brings Up An Object's *Viewer*

Every object's viewer shows all of the object's properties and behaviors organized into "categories" (or "traits" or "roles").

We can paint a little red car

... get its viewer

... show its handles

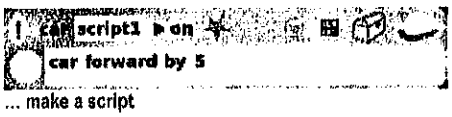


All Objects Are Scripted The Same Way

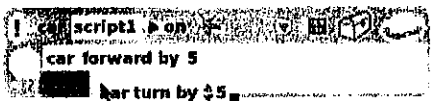
Scripts are made by dragging out tiles onto the desktop, and then dragging tiles into the script. Syntax is always correct.

... drag out some script tiles

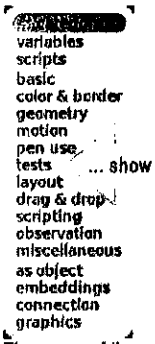
Tiles dragged from the viewer and dropped on the desktop ...



... make a script

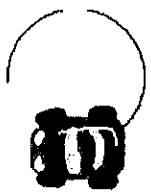


The green "destination marker" shows up when tiles are dragged over a script.

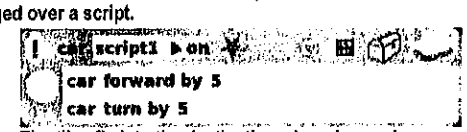


... show a role

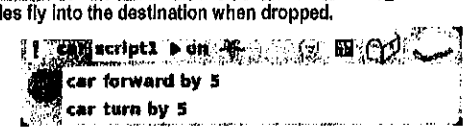
The menu of the standard roles



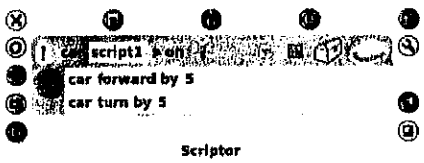
The car will move in a circle when the script ticks. If we have made "pen down" (in the "pen use" category) true, then the car will leave a trace as it moves.



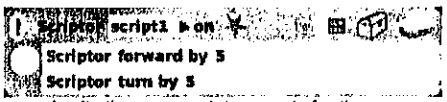
The tiles fly into the destination when dropped.



The script can be started "ticking" by clicking on the clock or by telling it to tick)



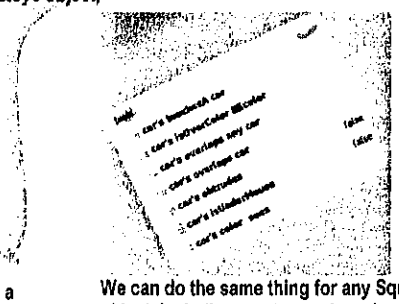
Because the script is a standard Squeak Etoys object, we can get its handles, and its viewer ...



... and write the same script we wrote for the car

Object viewer with some of the standard categories of properties and behaviors

The result is that the scriptor will move in a circle and leave a pen trail of its own



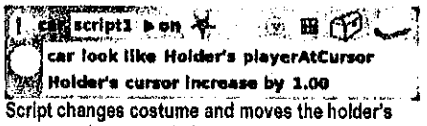
We can do the same thing for any Squeak object, including a category in a viewer.

Or a movie even while it is playing.

Animations Are Just Costume Changes ...

To change our car into an animated worm, we paint some worm costumes, drop them into a holder and write a script to move the cursor and change costumes.

Paintings in a holder

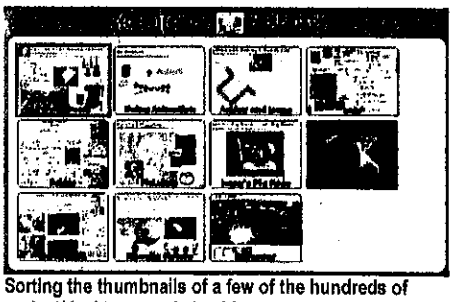


Script changes costume and moves the holder's cursor to the next painting.

... Movies and Videos Are Just Animations

Now we should realize that we have also made the guts of a movie player: we can just drop frames from the movie into a holder, write the two line script and it will play them.

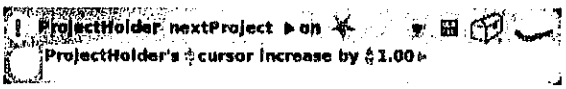
What's missing? Movies usually have hundreds to thousands of frames, each of which has lots of pixels. These frames are usually held as a file on the hard drive and are brought into the player application. Squeak Etoys provides automatic services for relating contents of files to Etoy objects, and also to compress and decompress pictures.

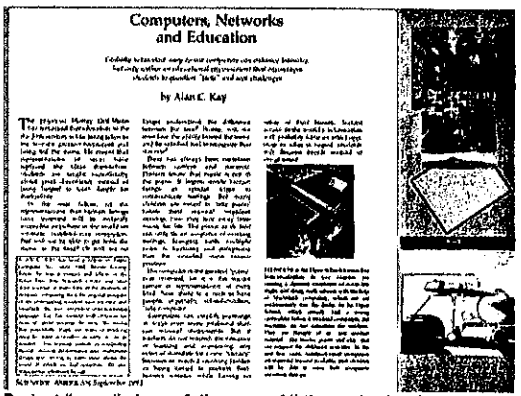


Sorting the thumbnails of a few of the hundreds of project/desktops made by this user

... Docs and Presentations Are Just "Animations"

The Etoy project/desktop is where things are made, and the end-user can have any number of them. They also act as "pages" for documents, presentations and the web. The pages are sortable by hand and by script, and each sorted sequence can be named and used – this allows many different presentations and "books" to be made up from the same materials. *These "pages" are just standard costumes on regular Etoy objects.* Squeak "Book" (multipage documents that are like a HyperCard stack) and Squeak Presentations (like PowerPoint but more powerful) are the very same kind of structure. The "pages" can be any object (including a whole project), and turning is done by a script that looks like:

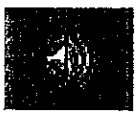


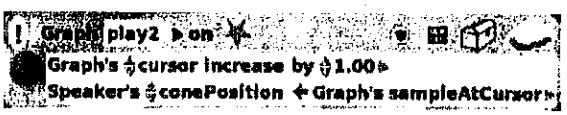


Project "page" shown full screen. All the authoring facilities are available, this is a live project, not just an image.

... Sound Synthesis Is Just An "Animation"

Now let's look at a different sequence in a holder. With the sound recorder we record a tone and see it is a sequence of bars, whose height indicates the sound pressure at that time.



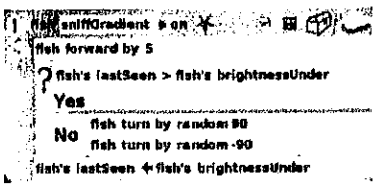


One of the objects supplied with Etoys looks like a loudspeaker and if we move this object, the physical loudspeaker will move. We write a little script that is like the animation scripts, but instead of doing a "looks like" we will move the loudspeaker according to the height of the current bar, and we hear the tone! If we change the "increase by" to 2, we will hear the tone an octave above, and if we change it to 1.5 we will hear the tone a fifth above. We have just made a "Model T" real-time synthesizer.

Massively-Parallel Typical Element Programming: "Particles" and "Fields"

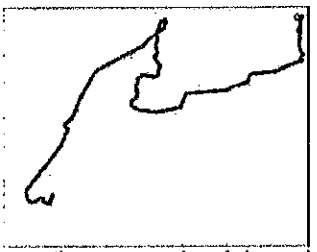
The metaphor of particles and fields can be applied to many ideas and situations – particularly given that the foundations of physics are couched in these terms, and thus "pretty much everything" can be represented this way.

A biological example starts with a salmon trying to swim upstream to its spawning grounds.



Virtually all biological organisms are able to follow gradients using almost no memory and an extremely simple (and very profound) strategy: if things are OK, keep going, otherwise tumble!

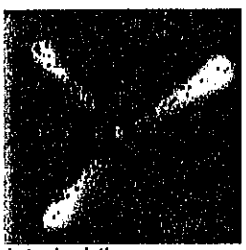
Here we have a "fish" looking for a more concentrated (darker) chemical. The gradient is not as smooth as it looks, so ...



... we see there are a number of places where tumbling took place. But the upper right corner was still found pretty efficiently

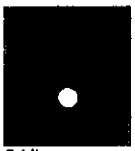
It's helpful for most learners to think through gradient following in the large with a single organism. But once done, there are many analogies with thousands and millions of particles that use gradient following for both efficiency and sending messages. For example, ants wander randomly looking for food. When they find it they carry it back to the nest and lay down a scent trail of chemicals that both diffuse and evaporate. If another ant that is foraging finds the scent trail it can follow it upwards towards the food source. The more interested ants the more scent and, if the scent evaporates slower than the supply, this means there is a lot of food and the bloom of scent will attract many ants.

From a systems perspective, this is a graphic and interesting example of loosely coupled coordination of massively parallel agents. The "field" is a virtual object, and can be represented by mathematical relations, message passing and events, or, in this case, the "field" diffusing and evaporation of the scent is done by 10s of thousands of stationary particles that form a background grid (they are essentially communicating finite automata). The Etoy system can potentially handle about one million graphical particles at 10 frames per second on a laptop.

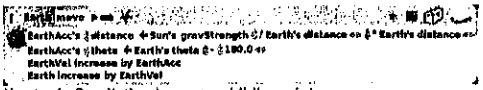


Ants simulation

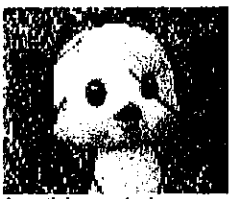
Gas particles raise a piston



Orbits



Newton's Gravitation Law as a child's script

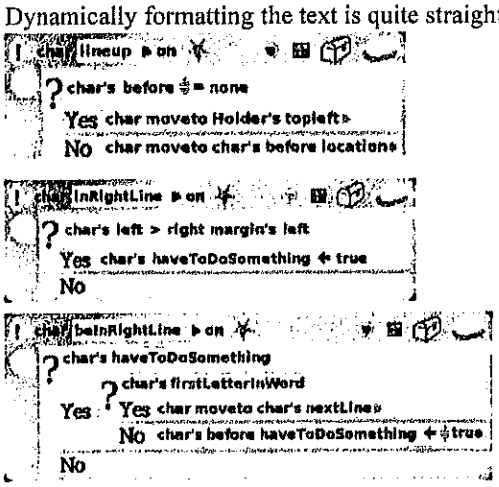


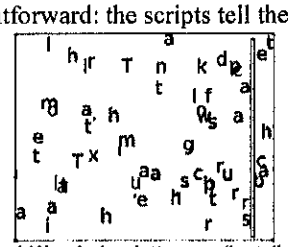
A particle per pixel

Massively parallel blurring

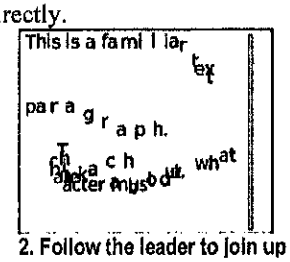
"Particle & Field" Text Paragraph in Etoys

Dynamically formatting the text is quite straightforward: the scripts tell the story directly.





1.Wandering letters as "ants"



2. Follow the leader to join up

This is a familiar text paragraph. Think about what each character must do.

3. All lined up

This is a familiar text paragraph. Think about what each character must do.

4. Moving margin causes text flow