Playground Essays

by Alan Kay

Preface

The Apple Vivarium is a long-range research program which uses children's interest in the forms and behavior of living things to inspire a wide variety of exploratory designs in curiculum, user interface, computer inputs and outputs, and modeling of behavior.

We expect that having children "learn about learning" and "think about thinking" will set up waves that lap the farthest shores of their minds—that trying to visualize the world through the senses of other creatures will give them a remarkable collection of perspectives from which to view their own world.

For us, the Vivarium is a Romance that demands new ideas and fresh approaches.

This document is a collection of essays written at different times about different parts of the Vivarium Program. They are still in draft form and were originally intended as design notes for internal use by the project members. The ATG Open House in June '88 is the target for them to be turned into a more coherent representation of our work and plans. The current set is not consistent but quite suggestive of the directions the program has taken.

Not all of the projects of the Vivarium are included here. Currently, the separate projects include:

The Open School
Playground
Physical & Mental Modeling
Computer Coach
Learning Seminars

This set of documents mainly discusses the first two. The remaining three areas are still in preparation.

This document contains Playground essays.

A Simulation Kit	-- Example
Salmon Spawning Behavior	-- Example
Dawkins' Biomorphs	-- Example
Adventure Games	-- Example
Traditional Animations Of Images & Sound	-- Examples
Electronic Mail	-- Example
A Paragraph Editor	-- Example
A Corner Mover	-- Example
A Scroll Bar	-- Example
Angular Momentum	-- Example
Playground History	

Playground as Medium

Though computer systems are sometimes portrayed metaphorically as "tools" or "vehicles", so many more dimensions and extensions are available that it makes more sense to think of "medium" as in "pencil and paper", "literary", and "artistic". What is the computer as a medium and how might it be shaped? Well, for one thing, its content is descriptions that can simulate all existing media and many more that can't directly exist in the physical world. Another level has been added to the idea of literacy. We are familiar with the first, that of fluency *within* a medium such as drawing and reading the printed word. The second level is fluency in the *shaping* of a useful medium from the metamaterial of the computer. Since children learn much about media by being embedded in it from birth—those who aren't are sorely hampered—it is very important to provide a user interface that is both eminently accessable and which gradually reveals the real nature of the powers available.

In fact, the presence of the children starts to "tell" us much about how the design should go. However, in order not to be led astray by simply providing an easy to use interface, what little is known about how the different mentalities respond to information must also be taken into account—in particular, the great differences between the kinds of thought that television and printing engender.

Bruner concentrated on three major mental processes that seemed to have rather different ways to deal with the world. Recently, models involving considerably more separate mentalities have been advanced. The important thing is that there is more than one, and almost certainly more than two. Just as central to the point is that none of the mentalities alone seems to be a good candidate to be amplified at the expense of the others.

As an example, visual logic is tremendously useful in getting creative activities started. The constant flitting around of attention allows many things to be considered without getting blocked—imagine how strange it would be if the first thing we saw in the day was all that we could look at for hours, but that is the way the symbolic "logical" mentality likes to work. So far so good. But visual logic also *implies* by spatial association. If someone were to assert in words that "eating a MacDonald's hamburger will turn you into a very attractive human being", a proposition has been formed that can be reflected on and easily rejected. But, as advertisers well know, we have no such negative reaction to images of handsome people eating brandname foods—the implication is "softer" but works all too well.

The visual mentality loves the new, and because of that prefers small scenes that

don't take long to solve, and is very interested when a scene changes. In theatre, this is sometimes called *spectacle*. The symbolic mentality on the other hand likes to mine the old for ever deeper insights and connections, but can get trapped and pedantic. Theatrically, this is called *substance*. Though we might prefer one to the other, it is fairly obvious that both working together can build much more interesting and deeper ideas than either in isolation.

Bruner's other mentality, the first to appear in a child's intellectual development, is one we scarcely consider. It is the mentality that deals with body logic—Bruner called it "enactive"—that not only orients us in the world, but *puts* us into the world as an actor. Thinking by acting, as young children do, is obviously very important, but can be very dangerous much of the time. Bruner believes that the other mentalities evolved to defer actions in the real and dangerous world to, first, an image based world in which choices can be considered and single stage next actions visualized, and second, to a logical world in which the vividness of the visual can be ignored in favor of long chained plans based on facts which may even contradict "common sense".

A happy synergy of Bruner's three mentalities would combine spontanaety, creativity, and thoughtful focus. Of course there is much more going on. The "three mentalities" might not even actually exist. We certainly know about more ways to appreciate the world than just these—music, for example, can with a struggle be mapped across the three mentalities, but it could just as well be a separate mental center itself. Just as there are a large number of ways that color vision could work—and nature happened to pick one of them—human psychology has not yet found which particular architecture generates our behavior. Be that as it may, design can get by without cold facts, but it can't survive without inspiration—and these multiple mentality ideas are enough on the right track to lead to fruitful ideas.

Thus our task in the design of a new computing system is first, to synergize as many mentalities as we can identify. Second, without cramping a particular mentality's style, to make sure that overshoot in a particular direction can be balanced from another. And finally to provide bridges from one to the other in accordance with the normal progression of development. By the latter, I mean that the very young child mostly thinks by doing, thus a bridge to the next stage—the visual—might be helpful. The elementary school child is predominantly visual, thus a bridge to the symbolic that can aid planning, and longer chains of inference, may be very helpful. It is important to reemphasize that there is nothing "wrong" about

the earlier stages—our goal is not to hurry the child to the next stage—but to start to supply the synergy amplification earlier. In fact, it is very likely that the most important aspect of the entire user interface design will be to recall the earlier mentalities after the strong dominence by the symbolic that starts to happen around age twelve.

These were some of the ideas behind the Xerox PARC "multiple window and pointing" interface that, refined by many talents, led to the Macintosh. The mouse is not there just because pointing is useful, but because gesture input connects to the enactive mentality which tells you where you are, that puts you *into* the world, and thus brings its world more strongly out to you. The icons and multiple windows are there to take advantage both of the efficiency of the iconic mentality and to provide a way to consider the creative possibilities without getting blocked. An underlying programming structure like HyperTalk or Smalltalk is provided to make a idea gotton by manipulation of hand and eye much more powerful through abstraction.

These ideas are still not well understood even by many who have worked on this style of interface, nor have any of us been able to carry all of the implications forward with equal balance. The two biggest shortfalls of these principles on the Mac have been, first, the lack of multiple windows in many applications, and in those that have them, to fail to provide multiple views of an underlying model. Thus, we have the strange case of one-windowed HyperCard, which when you want to use Help, forces you to abandon the very scene that you were puzzled about—nor can you compare two or more cards, even though comparison and contrasting of information is more important for school and elsewhere than retrieving an isolated fact. This shortfall is due to lack of understanding by implementers; techniques exist to do it correctly.

The second, and most important, shortfall is the hookup to the symbolic mentality via a kind of programming that extends outwards from the more concrete enactive and iconic activities. Hypercard currently does it best, at least along one dimension, and much can be learned from its enthusiastic acceptance—but it falls woefully short in too many places. Playground is another essay in this direction; it will not be a big surprise if it too falls woefully short. The history of programming language design is littered with brave failures—unfortunately, most of them are still being programmed in today. This shortfall is due to real lack of knowledge and ideas; techniques to accomplish it have been hard to come by, partly because implementing a programming system to do anything useful is such a large and slow task.

Playground is being designed for elementary aged children but is expected to extend far into the world of adults. At the most ambitious end of the scale, we want children to be able to build biologically sensible animal mentalities and spatial models. At the most mundane end of the scale, the children need dynamedia—sketching, drawing, animation, music, word processing, desktop publishing, information storage and retrieval, electronic mail, collaborative work and more. The teachers have to be able to understand the system in order to guide and answer questions. It must be very simple for both sets of users or it will never make its way into the classroom. It must be very comprehensive in its scope or getting it into the classroom will be an empty or even debilitative gesture.

Our approach has been to take the hardest thing we want children to do—programming an animal mentality and its spatial extension—try to make it simple enough to be possible, and then to see if the metaphors that arose could be used for more conventional programming and manipulation. Our first problem, of course, is that no adult had ever programmed an animal mentality above the level of a protozoan. Parts of higher animal mentalities had been simulated by early neural nets, but no higher level architectures emerged. Thus one of our subtasks has been to find an architecture that seems fruitful, build lots of models, then look for elegant simplifications. (Graduate students are great at this). Our current architecture is a combination of some beyond-Smalltalk object-oriented ideas and Marvin Minsky's Society of Mind.

Here is an example by Mike Travers, a Vivarium graduate student at MIT's Media Lab. The main creature being modeled is a three-spined stickleback, perhaps the most studied fish, first described in detail by Tinbergen in his classic: *The Study Of Instinct*. Much of what follows is motivated by Travers' paper **Animal Construction Kits.**

> To implement a real creature we must implement a realistic world for it to live in. The stickleback has to be able to sense the onset of mating season, sense the presence of other animals, including their social signals such as the red belly of the male, sense the presence of eggs in the nest and the actions of its offspring. If it is to dig its nest, the physics of digging must be simulated to some degree of detail.

And there is much much more in the stickleback's world.

Travers has built everything from a new kind of object called an *agent*. Everything is an agent, an agent is made from agents, and agents are spontaneously

active: they can *notice* their environment, many can run at the same time, and the intensity of their activity is controlled by an activation level. Agents can activate or suppress others including sensory and motor agents, remember the current activation state of other agents, and create a new agent or alter an existing one.

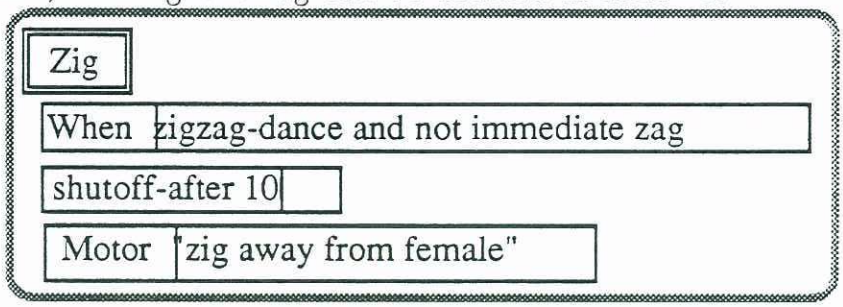
An example of a lower level agent for a stickleback would be one that can recognize sticklebacks. As with all agents it is composed of agents. This one seems quite complex but Tinbergen showed that there are many forms that will incite the same response particularly around mating time. Almost anything about the right size with a red belly will do. Where do the agents stop? This depends on the level of "physics" that is built into the environmental simulation. At one level much can be learned by having already made up agents for lower senses and motor activities. At another level it will be useful for the children to build a stickleback recognition agent from simpler ones. If our environmental simulation were on the equivalent of the E&S CT6 flight simulator, then the primitive agents might be size, shape, and color estimators. We can at least image that in the environment are certain "physics" agents such as *move*, that implements momentum, so that an object put in motion will tend to continue in motion; and *drag*, that implements frictional drag at a constant rate for each simulation cycle. Another physics agent would implement gravity, or more useful for the fish world, buoyancy.

Another lower level agent might be one that turns on male reproductive behavior. It could be as primitive as a simple test for its host being male and that the season of the year is "springtime"—the latter information supplied by the god of the machine or by an accumulation of still lower order sense information having to do with the passage of time, temperature of the water, condition of the light, etc. The general rule in the classroom would be that any area of interest would be simulated to the finest degree affordable by the computer, other areas can be grossly approximated until they go to center stage.

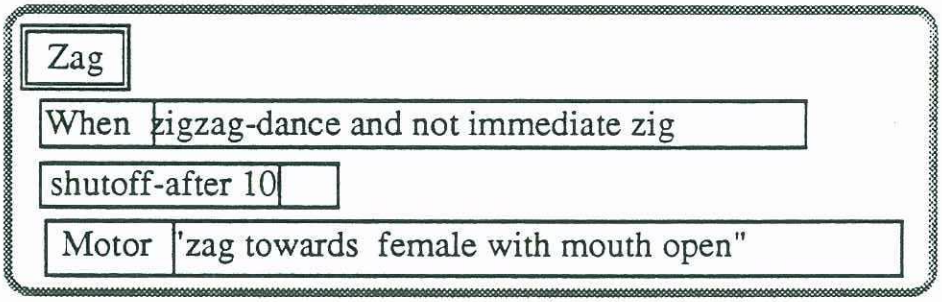
Another way to think of agents is that each one implements a *goal* or a *drive*. There are activation conditions that start and stop the goal and a level that controls the intensity of the achievement of the goal. There are subgoals made from activations of agents that may also be controlled by sensing conditions. This allows for more complex reactions than simple Skinneristic stimulus-and-response. A simple mentality when presented with food that it wishes to go towards and a predator that it wishes to avoid will go into fugue. A more advanced mentality will, in the most important cases anyway, have another agent that tries to resolve conflicts. Most animals when fleeing from a predator will not salivate if food is

placed under their nose. One way to intrepret this in the S-R world is as a kind of anesthesia. In a model that has drives it means that the drive that is in control only uses the sense agents it needs; everything else is simply not invoked.

Travers has supplied two useful "continuity" agents: *shutoff-after*, which ticks away and then shuts off the agent, and *immediate*, which tests to see what goal we were last trying to accomplish. In combination they can be used to produce oscillatory sequences. His example is, if *zigzag-dance* is a behavior caused by courting a female, then *zig* and *zag* can be defined as follows:



and



Travers comments:

> It should be noted that the hierarchical behavior structures of classical ethology are implicitely encoded here by the interagent references in *when* clauses, which provide a more general linking mechanism.

What follows in his paper is an very interesting discussion of how Minsky's "K-line" type learning and Schank "scripting" might be implemented using agents. But we have already seen enough to motivate the next discussion that concerns how this "ethological simulator" might be turned into a useful general purpose programming language.

We know from experience with the first completely object-oriented language, Smalltalk, that objects are a universal building block; everything can be made from them. Since agents are objects, we need only worry about the ease of programming and the understandability of the result. We have several main tasks. First, to make user interface actions a part of the language, and vice versa: so there is a direct

correspondence between the concrete and abstract. Second, to set up the evnvironment so that 80-90% of the actual programming is done by direct construction. Third, to have simple programs be obvious and simple. Fourth, to have the more complex already supplied agents—such as a MacParagraph—be themselves programmed in agents. *And* still be understandable—so that we can pop the hood" from most applications and not be dismayed or repulsed at what we find. Fifth, we have to build in all the facilities demanded by modern day computing power and network access—such as real-time animation and collaborative work. Finally, we must remember that our initial and most important clientele are children, and nothing of the design can be considered a success unless they can really use and grow with it—so nothing can be invisible and there must be no "work arounds".

Playground In A Nutshell

The initial conception of Playground was that of a "very large and extended drawing system (nowadays we would say MacDrawlike)"— a kind of Disneyland seen from above—in which everything lived, and every player-object had been constructed in a way that was also open to the end user. Part of the Playground would be common to all—events happening in the common would show in all the machines on the network instantly.

Everything in Playground is a player and made from players. We see a player on the screen as one of possibly many views of the player. The construction of a player is accomplished by making a view—every player that is moved into the view (ala MacDraw) becomes a component of the resulting prototype. There is a default automatic view that simply shows all of the components. Most views of a player will not show all of the components and those components that are visible will be in a graphic layout. Menus are simply another view that group some of the components that are sensitive to pointing actions. All of this implies that views are not as they are in Smalltalk (quite separate entities that are *applied* to models)—instead, Playground views are *owned* by the prototype and they are the only ways in which we can see and deal with a particular player. Editing the components of any view implies a possible change to the prototype.

A given instance is not restricted to just the components of its prototype—it can have individual players of its own for which the prototype doesn't have direct knowledge. At some point the abuse of this ability will cause kludgery, but Smalltalk's way is too restrictive.

The direct components of a player have names and they can refer to each other using those names. Some of the components will be public—outside players can refer to them by mentioning their owner and then their name. Private components can only be referred to by their own siblings.

Since each component is also a player what we see when we look at a component is one of its views.

A component that shows just one thing in its view can act very much like a spreadsheet cell if its internal goal is to find a suitable value to display.

The value-views of all the components are represented by the Playground system in such a way that any player can be found by searching on any of its internal values. Thus a player also acts very much like an active datarecord in a sophisticated database.

One of the simplest player-components is a button—the view is simply the name of the button—and the internals of the button are a goal to be carried out. Thus a button is much like a Smalltalk message-method, except that it can be actively looking for more than a simple message "push", as we see next.

In Smalltalk, we have to explicitly tell an object to notice an event (as though the object were poked with a finger to arouse it from a light slumber, or composed of buttons that need to be explicitly pushed)—and general broadcasting of messages is awkward. In Playground, individual players can notice much of what is going on around them (as though they have senses beyond direct touch such as smell, hearing and vision).

Playground is "event-guided" not "event-driven" in the following sense. A button in an event driven system (such as HyperCard) will have a module of code that (for example) responds to a "mouse-still-down" event. There is an ambiguity since there are several actions that might want to drive themselves from this event: such as (1) the button action itself is going to carry out some operation continuously as long as the mouse is still down—like scrolling something etc., or (2) we might be trying to do something to the button itself regardless of its intended action—like moving it to a different place, in which case we don't want to go off. HyperCard addresses this by walling off all button moving actions et. al. in a cursor mode so that the user can't deal with them at all—this is diametrically opposed to the philosophy of Playground. A more sophisticated event driven system might employ "flags" to indicate which goal is being pursued (none of the code examples in this note have necessarily anything directly to do with what Playground code will look like):

```
mouseStillDown
    if trying to moveSelf then ...
    if trying to doButtonAction then ...
    if ... etc.
```

This is better. But if biology is consulted, we get more guidance. It is a very weak "behaviorism" way of looking at things to think of a stimulus *causing* a response. Nowadays it is realized that there are goals (or drives, or what have you) already set up that *use* sense information to try to complete themselves. These goal modules compete with each other in such a way that a given stimulus does not always invoke

the same response—in fact, a goal may be so blocked that none of its usual response activity may happen at all. Thus we are led to the other way to organize which is "event-guided":

```
moveSelf Drive
    when moveGesture then ...
    ------------------------------
    when mouseStillDown then self location = mouse location
    ...
```

```
doButtonAction Drive
    when mouseDown then ...
    ------------------------------
    when mouseStillDown then ...
    ...
```

In other words, we group by goals and trigger by events. One way to disambiguate which goal is actually going to be served is to have a "guard" or "cue" section in the beginning of the goal—here anything above the "------" will trigger the whole goal. Once this happens, only the goal that has been triggered will be able to notice the sustained event "mouseStillDown". (If several goals are triggered, they will each be able to notice "mouseStillDown.) This has the same effect as mode flags but is conceptually and biologically more clear.

This ordering by subgoals that are attentive to events also resembles good documentation—we can imagine a manual saying:

> To move a button, first make a "move gesture" (by stroking to the left with the mouse) and then, while still holding the mouse down, move to its new location.

One of the biggest goals of Playground is that the code should be in one-for-one correspondance to user actions. To apply this to the Macintosh style of interaction, this—in part—means that many goals will be of the form

(1) have motivation for doing something,

(2) search-find-select a(some) player(s),

(3a) change the player's attributes or

(3b) replace the selection with other players.

Smalltalk conforms to (2 and 3) fairly well. The Smalltalk expression can be thought of as a retriever of objects, the cascades (using ";") allow many attribute

changes to be made on the result. The "3b" part only works for variable bindings and "from:to:"s in collections. In Playground we want the correspondance to be complete—there must be a motivation, a selection, and a change.

Goals will often be a sequence of subgoals.

when moveGesture | **then** (notice) mouseStillDown **then** self location = mouse location **then**

can be thought of as a sequence (ala Prolog and Parlog) where **then** is half sequencer and half **and**.

Goals serve the same purpose as Smalltalk methods and must not ever be more complex. Indeed, they should almost always be simpler to state.

An important use of goals beyond direct activities such as building things, seeking food, avoiding predators, etc., will be to notice conflicts between goals and help to resolve them. For example, most of the time goals can be quite independant—in most animals they are remarkably so. So a food-seeker can go after food. A sleeper can get the animal to sleep. A predator-avoider can get the animal to flee. When a steak is right in front of a predator we have conflict. Some animals can get trapped in an infinite approach avoidance loop. Most will have a conflict resolver that will gain control after some time to get the animal to do something else to resolve the problem. Monkeys have one for the steak and predator, but not for the nut in the jar—once they grab the nut and make their fist too big to get back out they are trapped by the strength of their food-seeker and the lack of a resolver.

Even though Prolog has a few wonderful ideas, its basic view of an unchanging universe with no time flow is not intuitive—indeed, it then has to introduce many ugly features to in fact change the universe, to control speculation, etc., so that the result is quite unesthetic. In Playground, we want the players to have state and to change their state as time progresses. But we have also introduced a nonprocedural way to find goals as contributors to a larger goal—and enough unsychronized parallism to make state changes dangerous and race conditions almost inevitable. In other words we have to deal with the concepts of "safe-unsafe", "undo", "backtracking" vs. "trial evaluation" etc.

We can make Playground a lot safer without the user ever being aware by dividing time flow for a player into two phases: (1) get to next state, I'm unstable so nobody can look (2) now I'm stable, allow others to see my state. This has to be done for general display, animation, and the network, and works well for general

computation. Each player now has a previous state that can be used to get to the new one with no race conditions on any values. This mechanism also allows for "slippage" viewing by other players, especially the displayer—it can work on a stable collection of player states while the players compute ahead without being lockstepped by the slower display process.

Pedagogically and biologically, the concept of "trial evaluation" is more pleasing than "backtracking". Dropping an egg on the floor requires considerable effort to erase and (from the egg's viewpoint anyway) can't be undone. Two year olds think by doing and have others to clean up and usually prevent their unpremeditated actions from killing them. Bruner points out that a great invention of nature—expecially for species that have few offspring—was the internalization of actions to symbolic renderings in which many possible activities can be tested without penalty. This is trial evaluation—we move a copy of the universe forward and let state changes happen to it while the real universe remains untouched until we actually do some thing in it—similar to backtracking internally but quite a different way to think about thinking. In addition it leads to "possible worlds" ways of using different contexts for mapping out problems which I think is also biologically relevant. We don't need the whole solution to this now, but eventually we will actually be building mentalities that create an internal model that is manipulated in order to get thinking to progress.

The User Interface

Since one of Playground's major goals is that 80-90% of anything a user wants to make can be accomplished by construction, just how the user interface is set up, and how new players fit into the user interface, is critical. In particular, the interaction differences that are walled off in different applications on the Mac must be reconciled—how to select & move, what double click means, and so forth. For the child and the teacher, uniformity is worth quite a lot—especially when the system is trying to be comprehensive, universal, and expansive. What follows are the initial—still untested—interaction suggestions.

How players are selected illustrates many of Playground's user interface principles. The most natural way to select something is simply to touch it with the mouse button down. If the player is a button, this won't work unless there is a non-button rim that can be touched without firing off the button. If the entire surface of a player is sensitive, then placing the cursor over a player for a second without a mousedown will select it and bring up its menu of options.

All well and good so far, but it is important that we now ask: just what are we to think is going on while we do this? We see an arrow that is controlled by the mouse, but what is it and what else can we control? The Gallery is a collection of buttons. each of which can be controlled by the mouse and make a player somewhere on the screen. The "first among equals" button has the selection arrow in it. If we select some other button, such as the rectangle maker, we will get a tiny rectangle controlled by the mouse that will make a new rectangle on the playground when a drawthough is done. After one of these, we could either automatically go back to the selection arrow, or we could have the rectangle maker stick to the mouse, ready for another drawthrough. This is actually a critical issue with points on both sides; each has several cases that can be very annoying. As of now, we choose to have the player-maker stick to the mouse, but a back and forth movement with mouseup will "shake off" whatever player-maker is stuck in favor of the selection arrow. Much of both drawing and painting is greatly aided by having this "semi-mode".

And it gives us a way to think about the selection arrow itself—as a selection-player-maker. In other words, when we have the arrow on the mouse, a mousedown or drawthrough makes a selection-player that, in a fairly flexible fashion, starts to enclose the players underneath. thus everything that is put into the gallery can be thought of in the same way: as a button that makes a player whose

location (and, often, extent) is controlled by a mousedown or drawthrough.

Now we have to consider the matter of embedded levels. A paragraph is a player whose contents are players acting as lines whose contents are players acting as text characters. When we touch the middle of the paragraph, do we mean to select the whole paragraph, a line, or a character?. What does a drawthrough mean? Well, most of the time we want it to mean select the characters for some actions, and incidently remember which line, and paragraph, and column, and page, and document you are in also. If this were adopted as the general rule, then we have several ways to get at the enclosing levels if that is what we really wanted. For example, anywhere we point in a paragraph that does not have a line or a character can unambiguously select the whole paragraph. Likewise, anywhere we point in a line that does not have a character can mean select the entire line. This can also be adopted as a general rule for any players which have views that show other players. More complicated situations can be handled by actions (such as going to a menu button) that progressively select more enclusive levels, or even by showing a "sideways" map that adds a third dimension to the embedment so that the correct enclosure can be selected—this will be an extremely rare occurence for most users.

A natural way to select several things is to draw through them. This raises questions about extension and movement.We want to do user interactions with just one-button mouse actions and no additional control keys. Since a selection is itself a player, it can have an *extension* birdie whose activation means that the next mouse actions are logically part of the selection. If the selection is larger than the window, then the pause method will invoke its menu which will have an entry for extension. Players need to respond to "undrawthrough" as well; any reversals of the selection gesture while the mouse is still down should unselect.

Movement is another action that needs to be done naturally without many levels of command. There is a major conflict between selection and movement of Finder and MacDraw kinds of players and the players acting as text characters in a text paragraph. To accomplish just a selection, the former requires empty space to start the selection drawthrough outside any of the players to be selected. This is not possible with jammed together text. A uniform way—whose annoyance factor must be tested—would be to put a *move* birdie on the selection and to place the cursor there automatically after the section has been accomplished. A move would then be a slow double click. If the cursor is shown just outside the selection while the move is going on, then the exact place where the movees should be dropped can easily be indicated with a mouseup.

One of the most useful views of a player will be an icon that shows the player in a closed up form. There are two major kinds of "openings" that need to be commanded. The first is the familiar "MacOpen" that shows the *role* the player has assumed—as a document, a card, an image, and so forth. The second is to see the player as the "entity under the costume" showing the scripts and the costumes that create the role.

Thus we see that the conception of Playground is much like a "very large and extended MacDrawlike drawing system—a kind of Disneyland seen from above—in which everything lived, and every player-object had been constructed in a way that was also open to the end user. Part of the Playground is common to all—events happening in the common shows in all the machines on the network instantly.

Everything in Playground is a player and made from players. We see a player on the screen as one of possibly many views of the player. The construction of a player is accomplished by making a view—every player that is moved into the view (ala MacDraw) becomes a component of the resulting prototype. There is a default automatic view that simply shows all of the components. Most views of a player will not show all of the components and those components that are visible will be in a graphic layout. Menus are simply another view that group some of the components that are sensitive to pointing actions. All of this implies that views are not as they are in Smalltalk (quite separate entities that are *applied* to models)—instead, Playground views are *owned* by the prototype and they are the only ways in which we can see and deal with a particular player. Editing the components of any view implies a possible change to the prototype.

A given instance is not restricted to just the components of its prototype—it can have individual players of its own for which the prototype doesn't have direct knowledge.

The direct components of a player have names and they can refer to each other using those names. Some of the components will be public—outside players can refer to them by mentioning their owner and then their name. Private components can only be referred to by their own siblings.

Since each component is also a player what we see when we look at a component is one of its views.

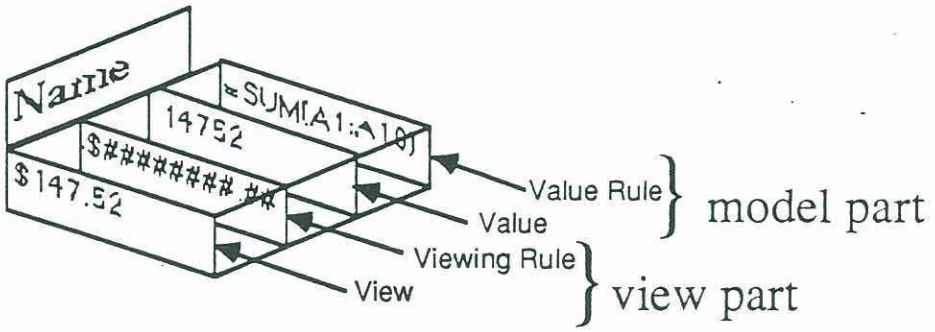
A component that shows just one thing in its view can act very much like a spreadsheet cell if its internal goal is to find a suitable value to display.

The value-views of all the components are represented by the Playground system in such a way that any player can be found by searching on any of its internal values. Thus a player also acts very much like an active datarecord in a sophisticated database.

One of the simplest player-components is a button—the view is simply the name of the button—and the internals of the button are a goal to be carried out. Thus a button is much like a Smalltalk message-method, except that it can be actively looking for more than a simple message "push".

Players and Views

The simplest way to think about Playground is that it is object-oriented, the objects are made from objects, and each object is a generalization of a spread-sheet cell.[1] We call the objects players (or agents) because they are both active and there are enough differences between them and classical objects to warrent a new term.[2] Interaction is WYSIWYG to a close a degree as possible—in particular, this means that an appearance of "partness" almost always means there is a part, and that a multiple viewing of the same player, the parts in each view indicate underlying players.



A Classical Spreadsheet Cell

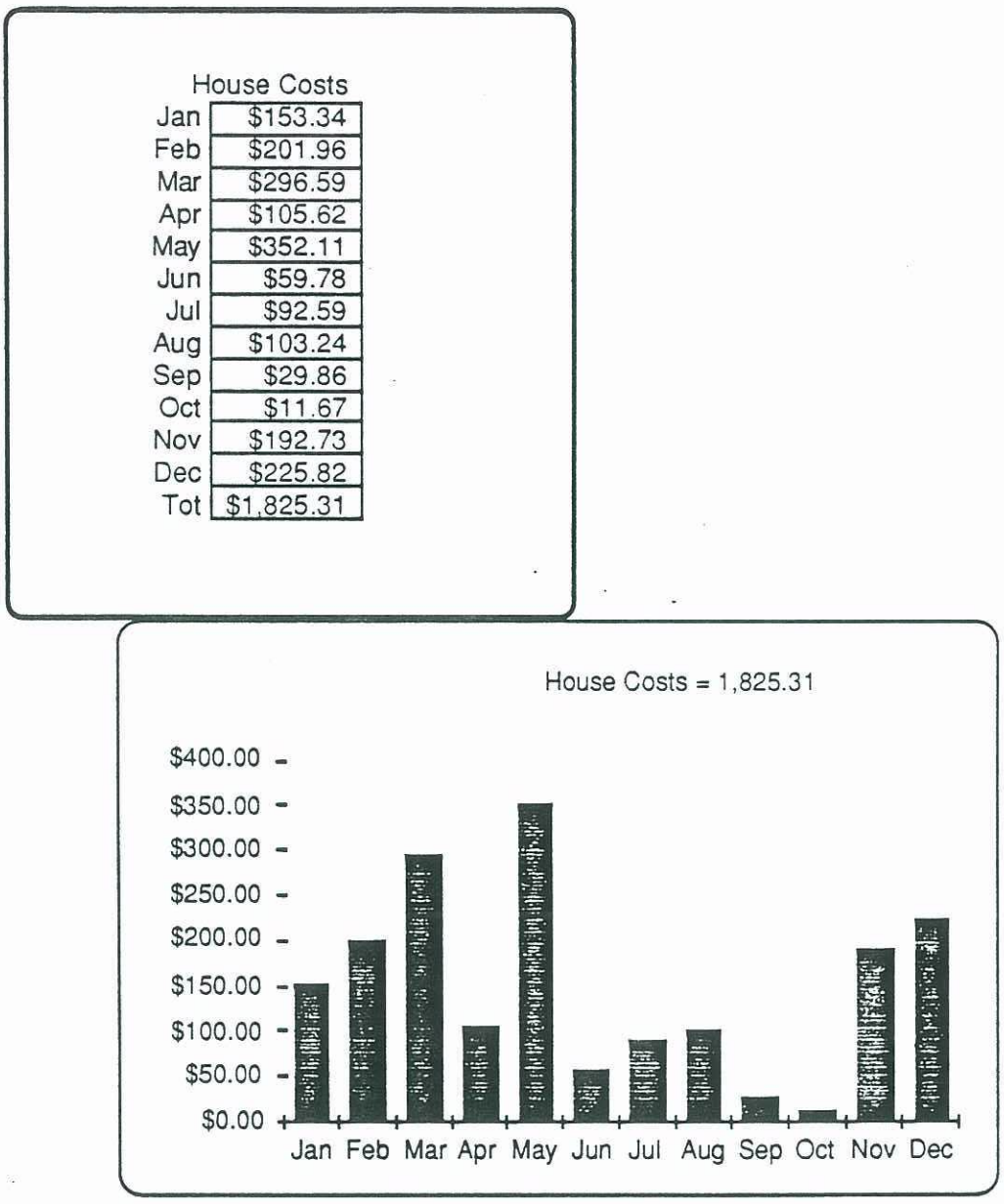
Before discussing Playground in detail, let me make some generalizations of the spreadsheet metaphor to motivate the later ideas. Suppose we allow the cells to be moved about individually within a larger container. In order to refer to cells from within a cell, we have to extend the ways of naming and refering. We can give each cell a name local to its container:, "income", "car-payment", etc. We can have a default name that lets us find all the cells in a container: "cell-1", "cell-53", etc. We can give a grouping of cells a name: "monthly-payments", etc. A cell can be in more than one group. A group can have structure of its own; one of these might be the familiar spreadsheet 2D grid.

Now let us manipulate the placement of the cells using MacDraw techniques. One useful layout would be to have a vertical column of house maintainence costs with a total at the bottom. Another would be a horizontal layout with the bar view turned on and the cell rotated.

---

[1]Early descriptions of Playground's biological "Tissue of cells" architecture appeared in "Computer Software" (*Scientific American*, Sept., 1984) and in "Opening The Hood Of A Word Processor" (*Apple*, Oct., 1984).

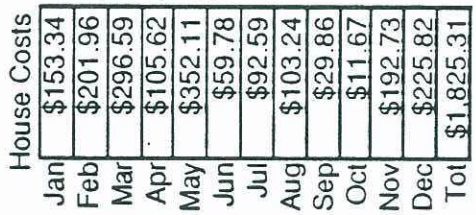
[2]The term "player" was first used to describe objects in "Programming Your Own Computer" (*Science Year*, World Book Encyclopedia, 1979).



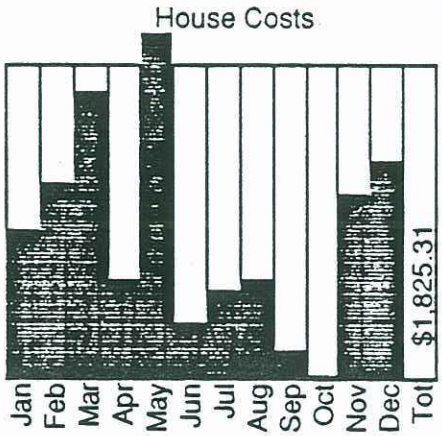
Each view is easy to build given our layout generalizations, but how easy is the programming as compared to an ordinary spreadsheet? What if the system allows us to do the following. We get a cell from the tool box, ala MacDraw and place it in the view. We type in the first number as though it were an ordinary spreadsheet cell or a MacParagraph. The convention in paragraphs is that a <return> means make a new paragraph, only show a label field if the current pargraph has one, position it below the current one, and put the cursor in it. This convention works very well for our spreadsheet cells. We can easily type in all the numbers; the labels are even easier to do than on an SS. The total field is done the same way as on an SS: we say "sum of" followed by a selection of the relevant cells; the simplest retrieval expression that collects all the cells is "cell-1 to cell-12" (a more complex one that

would be employed if the cell names weren't contiguous would be "{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}"). If we collected the 12 months and the total into a group, then the individual cell's generic names would be relative to the group, and references by outside cells would be changed automatically to make them relative to the group's name.

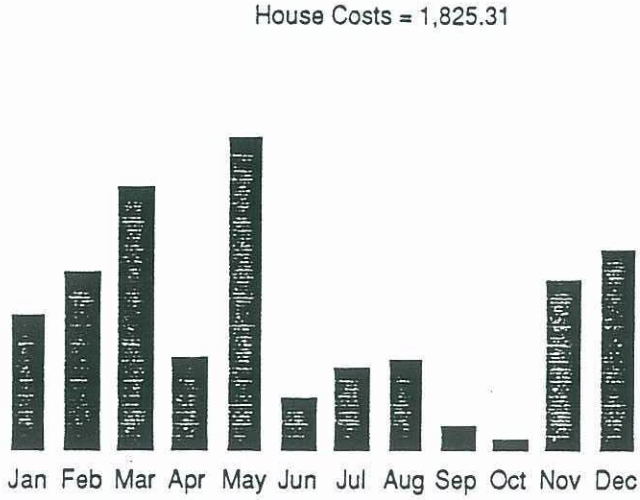
One of the ways to make a second view is to "copy" the existing view and make changes to it (we will see what the quote marks about "copy" mean in a moment). First we rotate the list of cells.



Next, we move the title to the top and change the view rule in the cells to be "bar".

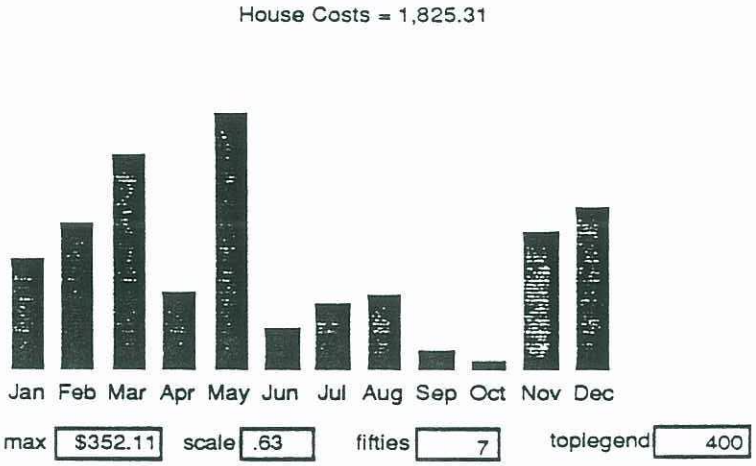


We decide to move the total up to the title, spread the bars, turn off the cell outlines and rotate the labels.



Now we have a choice as to how the legend that allows us to estimate the amounts will be entered. The easy way is just to draw it in by hand and have no dynamic connection between the legend magnitudes and the heights of the bars. A nicer way would be to calculate the legend magnitudes from the bars using a few more cells.

We make a cell called max that calculates the largest amount in the months: max=maximum of cell-1 to cell-12. Something that will occur to us later is that this calculation would be nice and general if we had grouped the cells into "amounts" so we could say: max=maximum of cell-first to cell-last of amounts. We want the maximum height of the bars to be about 75% of the view height: scale=(.75*view-height)/max. Now we go to the bar image rule and have it get its scaling from scale. All the bars scale correctly. To calculate the legend, we have to find the next higher "50" from max and then calculate the tic marks. Now we realize that the legend is actually going to be a bit higher than the bars, but we won't worry about that now. fifties=max//50 and toplegend=(1+fifties)*50.

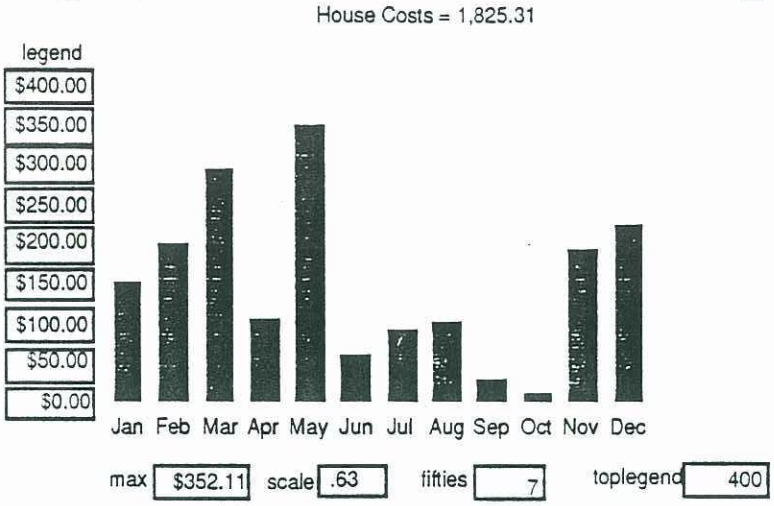


The next part is a little tricky because, to make it perfectly general, we have to create just the right number of cells that there are divisions of fifty. This is our second example of players that are created just for this view and are not part of the underlying model. To do it we write down what we would do if it had to be done by hand:

```
legend=repeat, with counter starting at 0, count to fifties+1,
        do Make new cell
            =toplegend-(50*counter)
                place at base widths of amounts - 50, base height of amounts + toplegend-(50*counter)
        end repeat
```

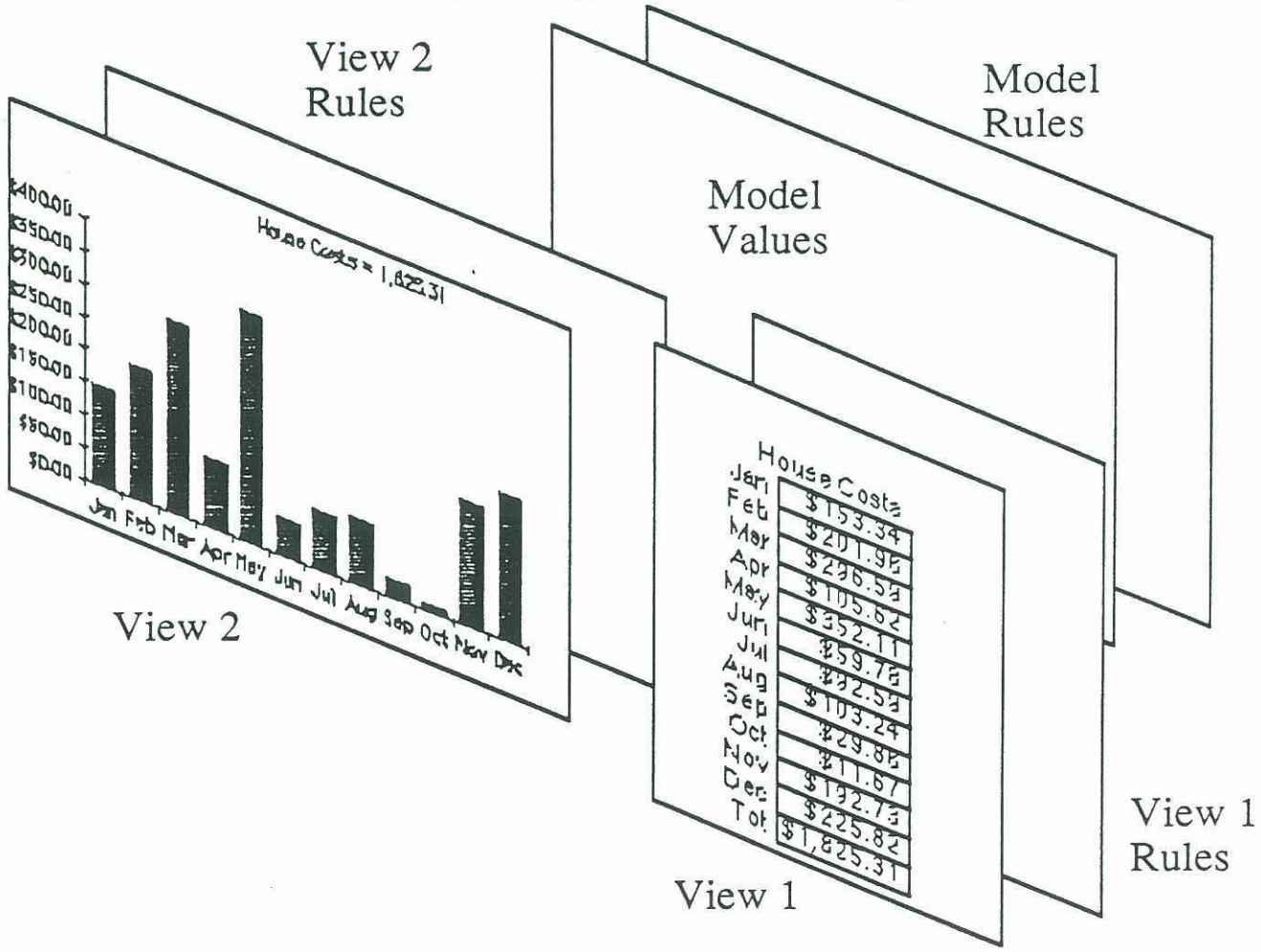
This makes nine new cells with the proper values, located in the proper place. Are these cells "real"? Can they be edited? Yes on both accounts. The cells are the values of legend and constitute a named group like amounts. Though they can be

edited, the entire group will be recreated whenever fifties changes.



A few more phrases in the legend script will get rid of the outlines and draw the tick marks in the proper place. All that remains is to make the label "legend" and the four auxillary cells invisible and the view is complete.

What we have created is simply a larger version of the simplest kind of cell.



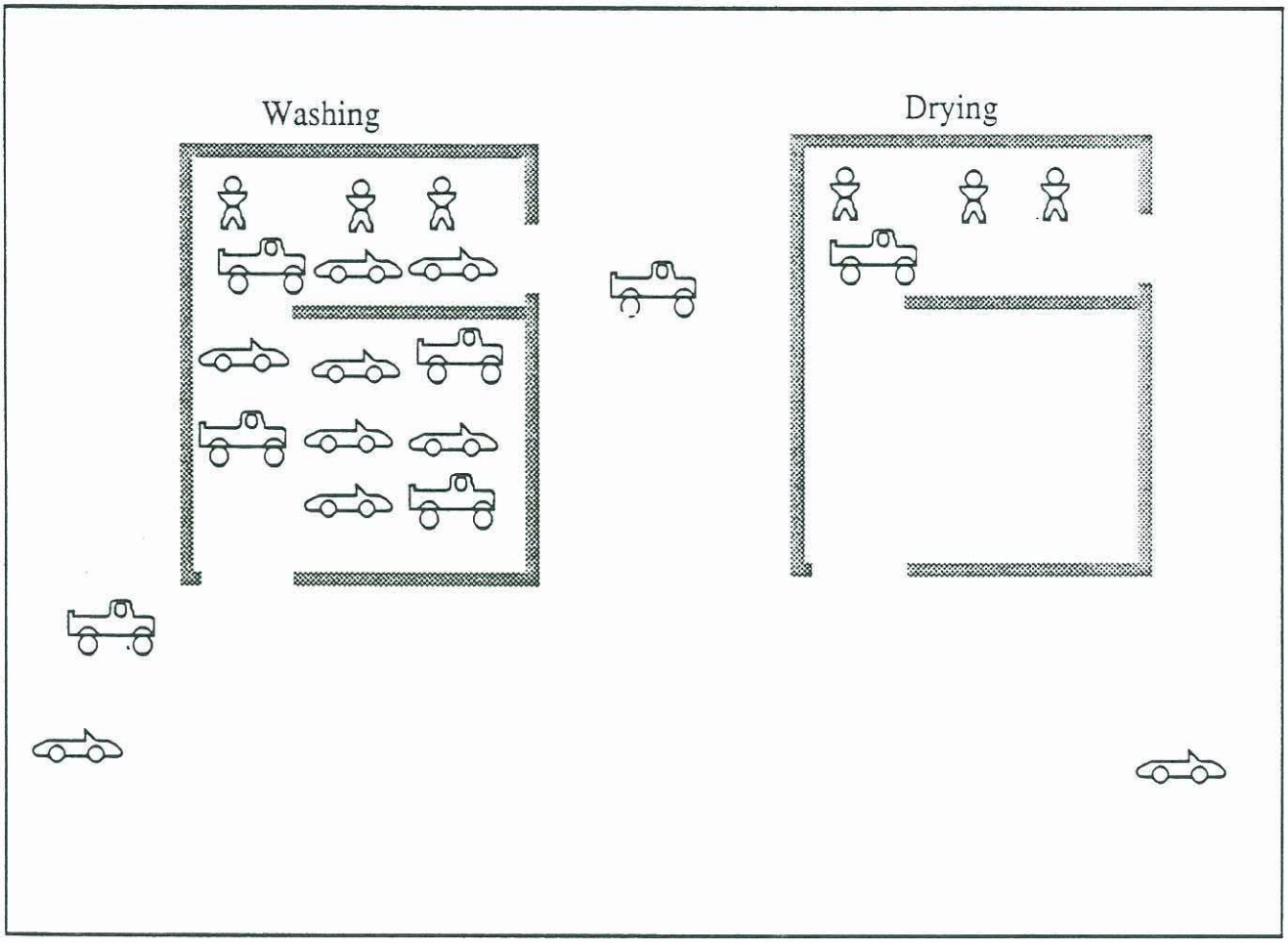
The model part is shared by its views. The value-rules and values of the cells are kept

in the model—to have them be different would be to disconnect objective reality. Each subview is obtained from the repetoire of the cell that had the value. This keeps the viewing model tractable and understandable. It is very important that most views created by users can be derived directly from a layout of the views of each part. However, a new door has been opened by the way the bar view was created. All its view rules expect is to see a model called House Cost that has a total cell and an amounts group. A little more work could make the view rules be able to lay out any number of months. If the model were changed slightly, so that House Cost was itself in a cell, then we would have created a general histogram viewer that could be applied to any model that had the correctly named cells. This is very useful because views take a long time to do. On the other hand, because of their generality, views can be very hard to understand. If most views can be easily made by hand then user overhead need only be expended when a very complex viewer that already exists is needed.

Our general rule for beginners and novices is that most things need to be buildable or copyable from scratch with very few components and fewer principles. When something really complex is needed, then there will be motivation to see if someone has already done it.

A Simulation Kit

Simulations are used to study systems that are difficult or impossible to deal with in the real world and have no easy analytic solution. Even the seemingly simple situation of a car wash for cars and trucks with a number of service bays and personnel requires simulation in order to understand how to staff the different areas for most efficient flow.



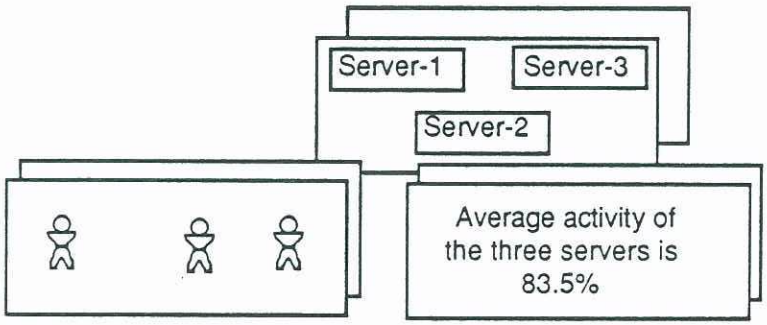
Another example of this type that children understand very well is an amusement park with rides, food concessions, and lines on which one must wait.This kind of simulation is called "A Job Shop" and consists of four main kinds of players:

- Jobs that need to be serviced in some way. A *car* needs to be washed and then dried. A *child* needs to go on several rides, then get some food, then go on more rides. Jobs usually have a schedule that gives them a plan for traveling.

- Servers that can give service to a job. A *car washer* is a person or machine that can wash a car. A *gondola* in a ride will take the child through the ride. A*waiter* in a restaurant will bring food. A *bed* in a hospital will provide convolescence. Servers usually have an average service time for providing service ad may have a schedule as well.
- Stations that provide a place in which the jobs and servers get assigned to each other. A *wash-bay* is a station where cars and trucks can get washed. Ohter stations are*rides*, *restaurants*, *hospital-depts*, etc. Stations have an entry and exit, a line that can be waited on, servers, and a place where the servers and jobs meet.
- Simulations that hold all the jobs, servers, and stations for a given simulation. Examples are: The Car Wash, The Amusement Park, The Restaurant, The Hospital.

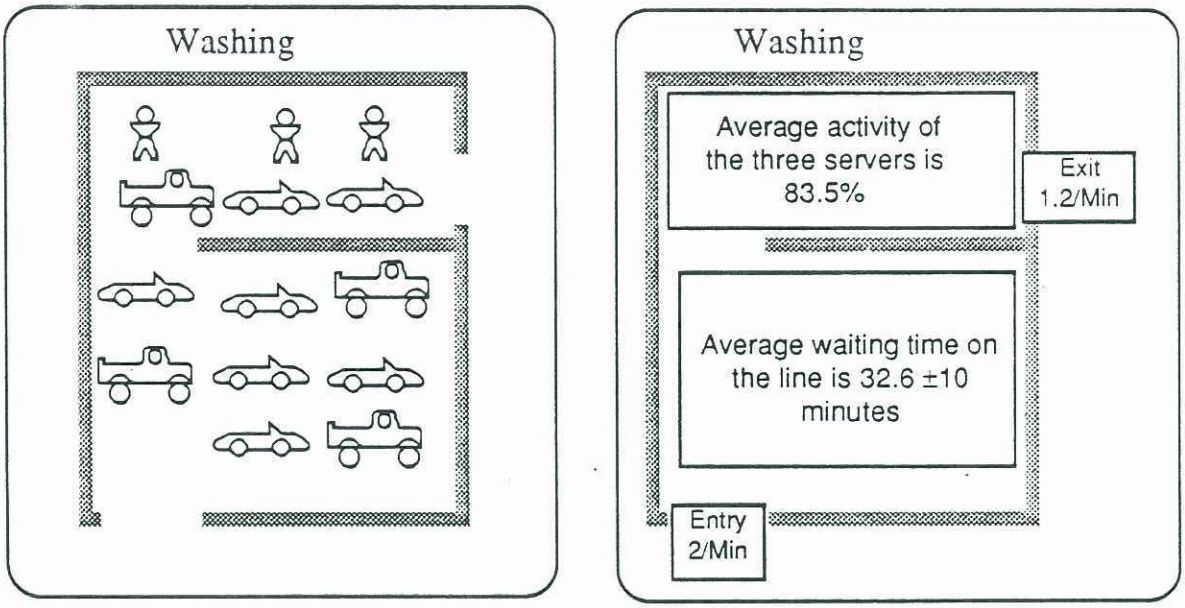
A *report* is just a kind of view. The intuitive view is one that is composed of icons and gives the user an immediate qualitative feel for what is going on—a waiting line is too long, some servers are too idle.

For example, the standard view of a server shows an iconic representation of a person, or gondola or bed. Another view that might be useful would be to compute a running average of a server's activity. A composite view would be the average activity of the group of servers.



The same principle applies to the waiting line. In one view we want to see the objects that are waiting; in another we might like to know the average waiting time spent in the line. The entry and exit areas also can have multiple views: a useful alternate would be to count the entries per minute so they can be compared with the exits per minute.

An alternate view of a station would simply combine the alternate views of its parts to yield a Station Report View.



The initial version of the simulation kit concentrates on the *meaning* of the simulation and ignores for the moment some of the niceties, such as how a job calculates a route from station to station without bumping into things.

A *Job* is a standard player that contains a *schedule* consisting of a list of stations to visit. Jobs are created by the simulation player according to its entry rule. Once put into the simulation, a job follows its schedule and then exits. A very simple travel method would be to look at the current item on the schedule, head toward it, and move forward a few steps.

```
travel=
    when I'm not in line and not at my schedule's 1st item's Entry's location
    set my heading to my schedule's 1st item's Entry's location
    move me forward 3 steps
```

Once the job gets to an entry, it can put itself on the waiting line for service. It won't be clear to the user yet as to when the schedule item should be discarded; but this is the first reasonable place. In the simplest form of the simulation kit, the job can just go to sleep until it is serviced and awakened at the exit for further travel. A more elaborate scheme would have the job awake—and perhaps worrying about other things it has to accomplish—thus the job might decide to get off the waiting line and try another station before coming back to the current crowded one. In this case it would be well for the job to remember all of its schedule until it actually gets service. The amount of pondering that a job might engage in extends all the way into the domain of expert systems.

The *waiting line* can be practically passive. Since a job has all the iconic characteristics of a text character, a line need only be able to hold text characters and put them on one end and take them off the other. One of these will already be in the tool box.

When a *server* is idle it can look at the waiting line to see if anything is in it—if so, then it can move the job near itself and then hold it for the server's average service time. Then it can move the job to the exit and wake it up.

```
service=
   when I'm idle and  line is not empty
   get first of line
   move it to my service area
   pause for my service time
   move it to exit
   give it to exit
   set my state to idle
```

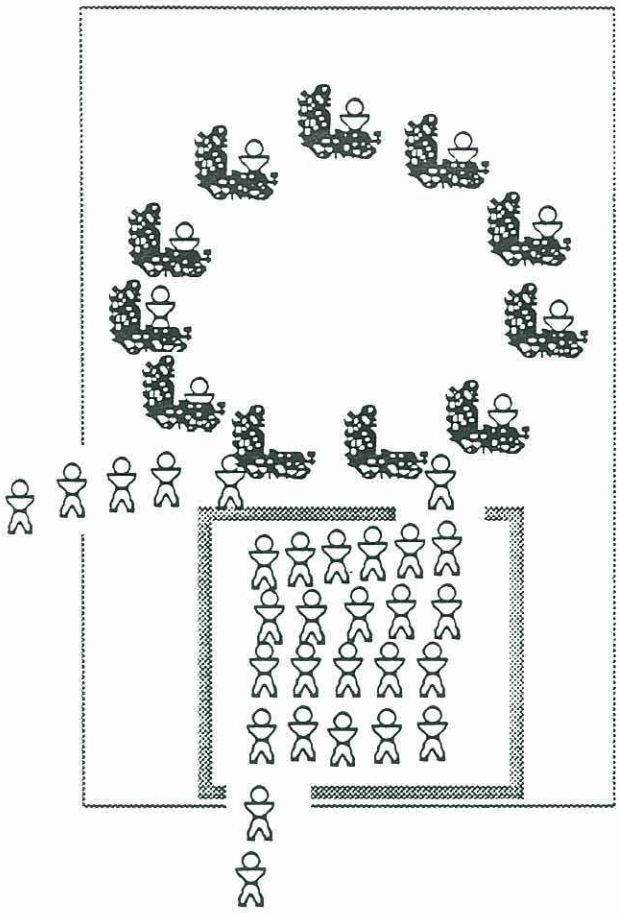
We see a good example here of having the behavior modules be *players that notice* as opposed to event-driven. There are other players that are also triggered by the idle and not idle events—such as the players that compute average activity.

```
average activity =
   set value to active+(active+notActive)

active =
   when not idle
   set value to value + 1

notActive
   when idle
   set value to value + 1
```

The active and notActive players are simply counters that count elementary ticks according to the state of their noticer. The noticing paradigm allows this activity to be separated from the generic server activity.

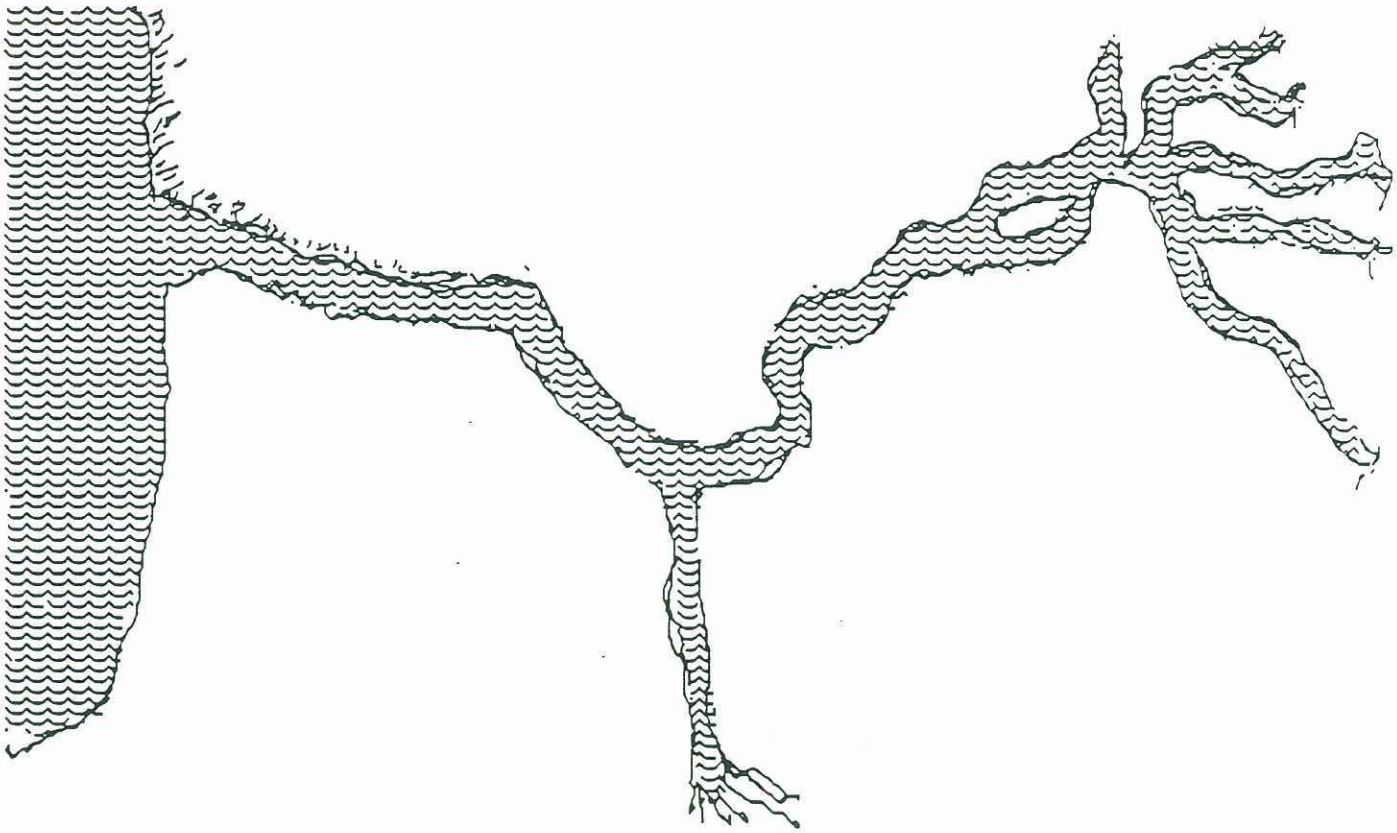
When the children simulate an amusement park, it will be important to provide more things to do when the servers encounter a job. Children, after all, are not primarily interested in the waiting line aspects of the amusement park. Playground's ability to move an object along a *path* can be used to layout abitrarily complex routes for the server gondolas to take.

Ferris Wheel



Salmon Spawning Behavior

We start with a large scale drawing of the ocean and an salmon river such as the Columbia.



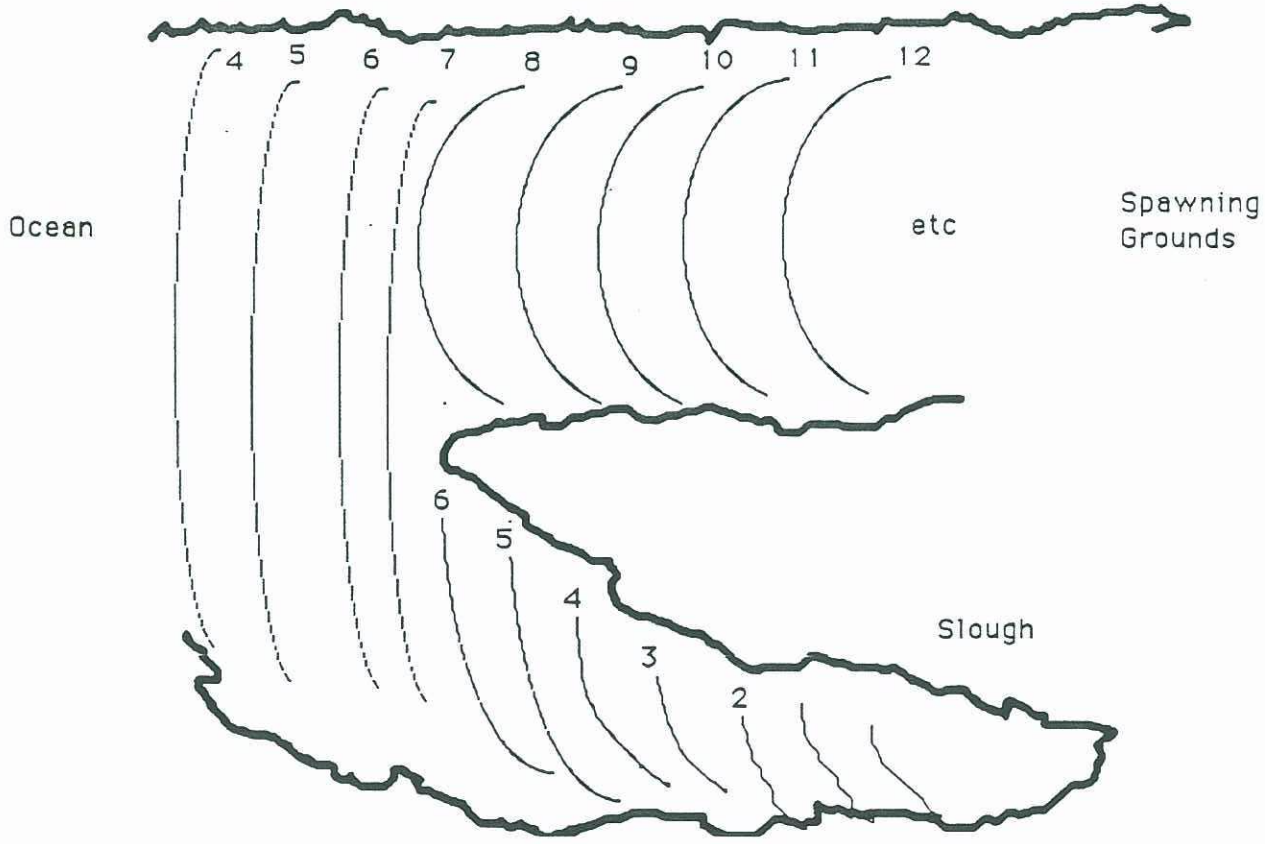
The children have already made some of Mike Travers' sticklebacks in a previous project. These can be placed in the ocean with their new salmon skins on where they happily swim around after food as did the sticklebacks. The physics players that set up momentum and drag are also active here. The banks are drawn with a particular paint that can be sensed by the salmon-players. We set up a new physics player whose job it is to "feel" the shoreline and prevent the fish from going through onto land—this player also sends a message to the "bump" player-sensor in the fish when contact happens.

When the salmon's season-sensor-player says "it is time to go back to the birth place to spawn", and very sensitive smell sensor starts to pick up gradients in the perfume given off by the specific place up river where the salmon was born. The salmon starts to follow those gradients with amazingly resolute purpose.

The players in the salmon that do the navigation can be quite simple:

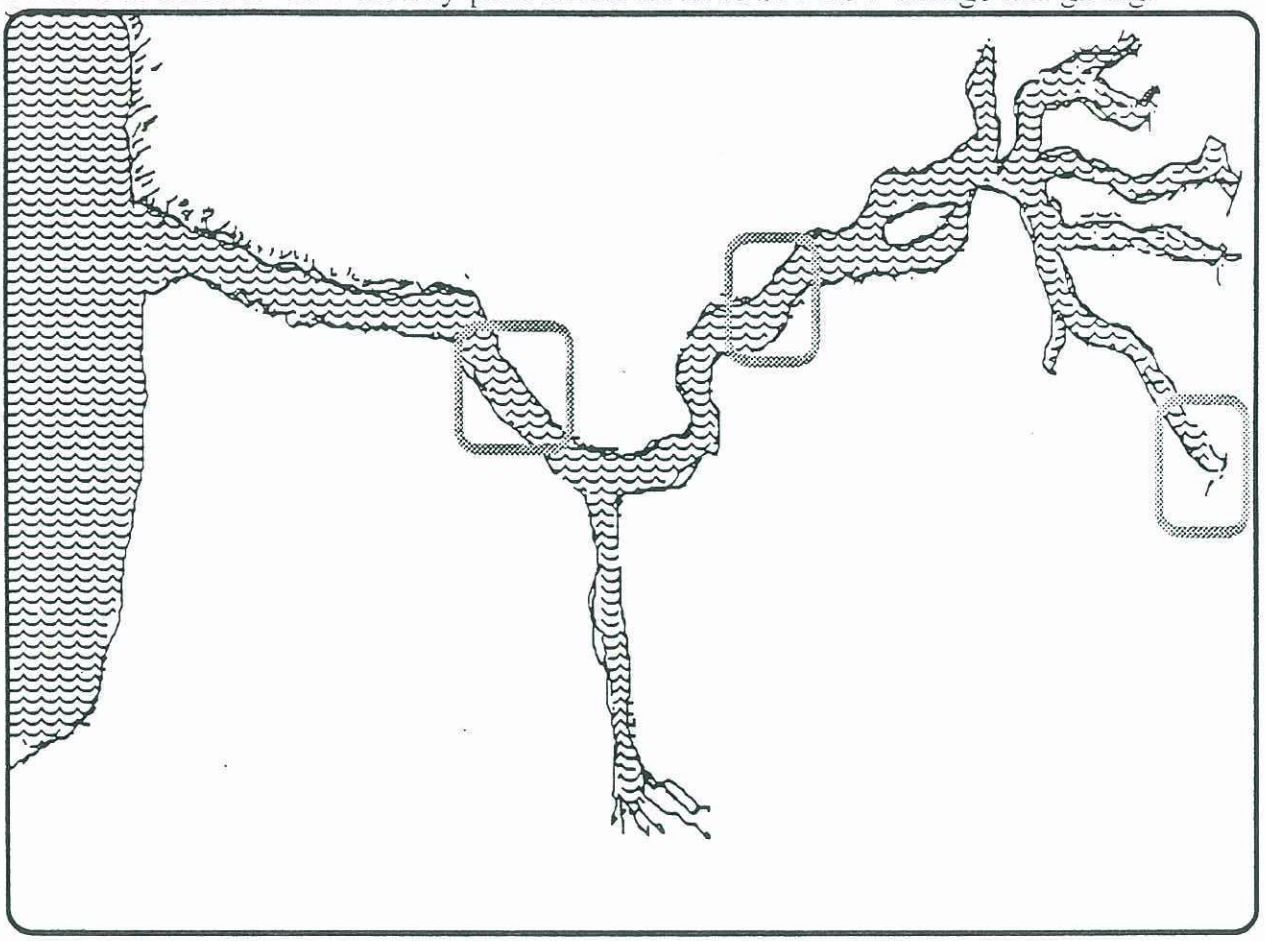
```
CurrentSmell =

PreviousSmell =

Turn A Little =
        when CurrentSmell roughly equal to PreviousSmell
        right 5

Turn A Lot =
        when CurrentSmell noticably less than PreviousSmell
        right 180
```

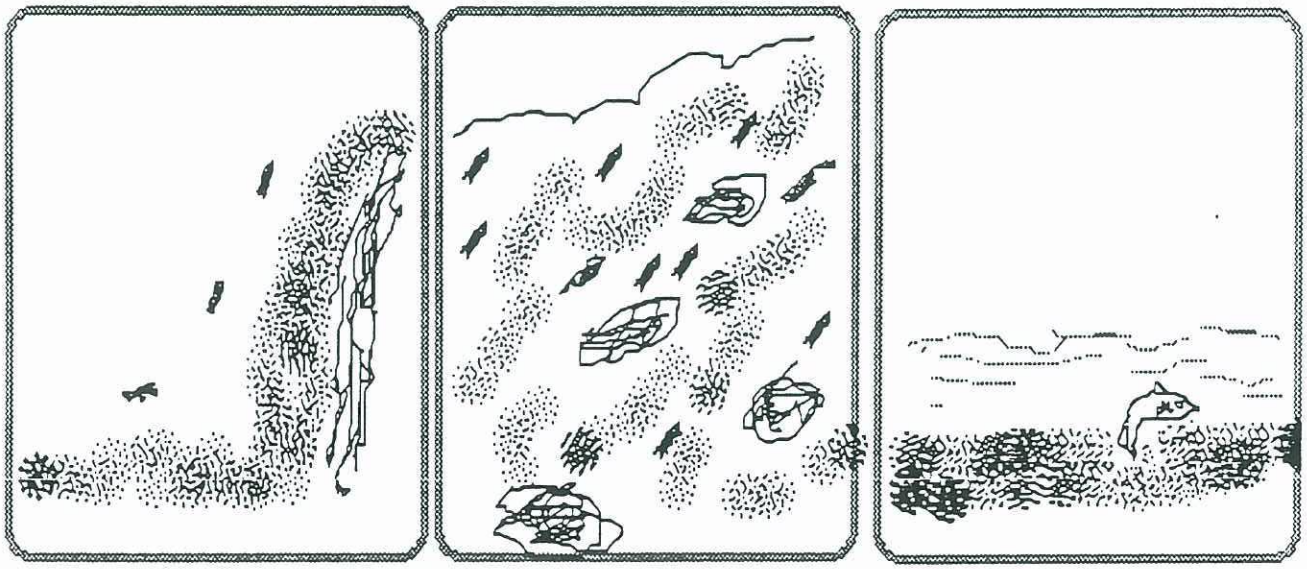
The perfume can be a bit misleading at times, as when some of it diffuses up a slough. I wonder how many salmon get trapped in sloughs, or do they have some other mechanism to resolve this problem?



Now, how to implement the perfume? One of the ways to do it is to think of it as a (usually) invisible paint whose color is the particular perfume and the brightness is inversly proportional to the distance from the source. Another way is to have a "smell line", a line-shaped player that is drawn from the spawning grounds down through the river to the ocean. The salmon's lower level sensors can notice the smell line and ask "how far?" and get an answer in terms of a perfume level.

Multiple views make the simulation much more interesting to watch. We want to be able to stick a view on any part of the river to see how things are going.



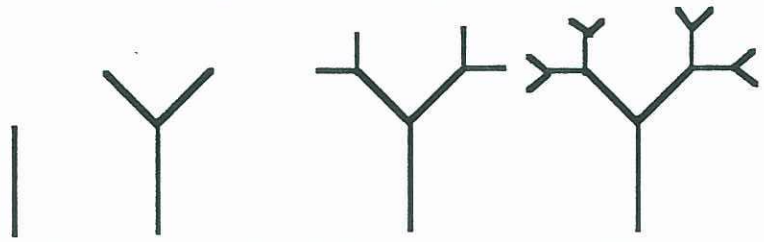


At the model level the salmon will be performing the actions as though it were in three dimensions, but because Playground is only "2-1/2 D"right now (and will be for the next several years) we have to set up separate animations for the side views of going up the falls and laying the eggs.

Dawkins' Biomorphs

While experimenting with a simple recursive tree drawing program, the evolutionary biologist Richard Dawkins discovered that, simply by changing the parameters more than is usually done, that a striking variety of shapes was generated. Soon, the now famous Biomorph Program was born.

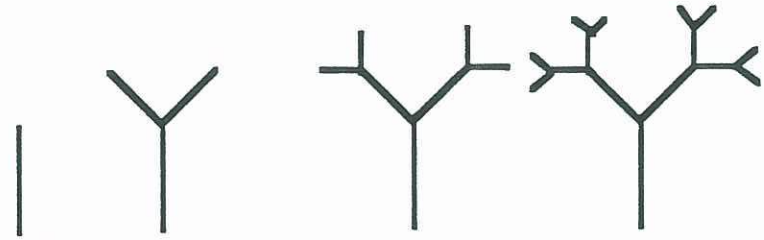
Here is how a fifth or sixth grader might program a biomorph generator. First, they have to see that a tree is composed of trees. One way to describe a tree:



is that it consists of a branch possibly followed by a smaller tree to the right and a smaller tree to the left. However, it is not at all obvious that the description is this compact. One good way to do it with children is to get "tree drawers", organize them and get them to draw sample trees.

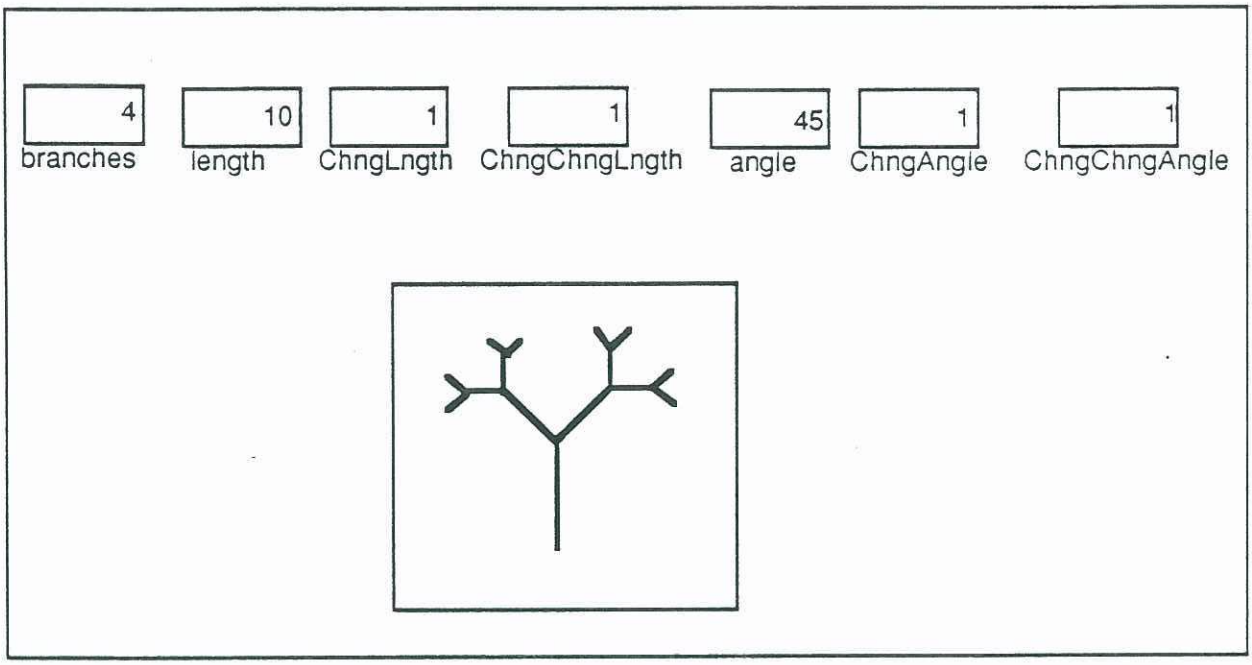
The program resembles the description:

```
tree branches length angle =
        if branches = 0 then quit
        forward length                          -- make a branch
        right angle                             -- turn to the right
        tree branches-1 length*.8 angle         -- make a smaller tree
        left 2*angle                            -- turn to the left
        tree branches-1 length*.8 angle         -- make a smaller tree
        right angle                             -- turtle to original angle
        back length                             -- turtle to original position
end tree
```



tree 1 10 45 tree 2 10 45 tree 3 10 45 tree 4 10 45

The parameters branches, length and angle can be thought of as genes. To get more variation, we can have genes that modify genes. We add a multiplier of the length and the angle, and a multiplier for the multipliers.

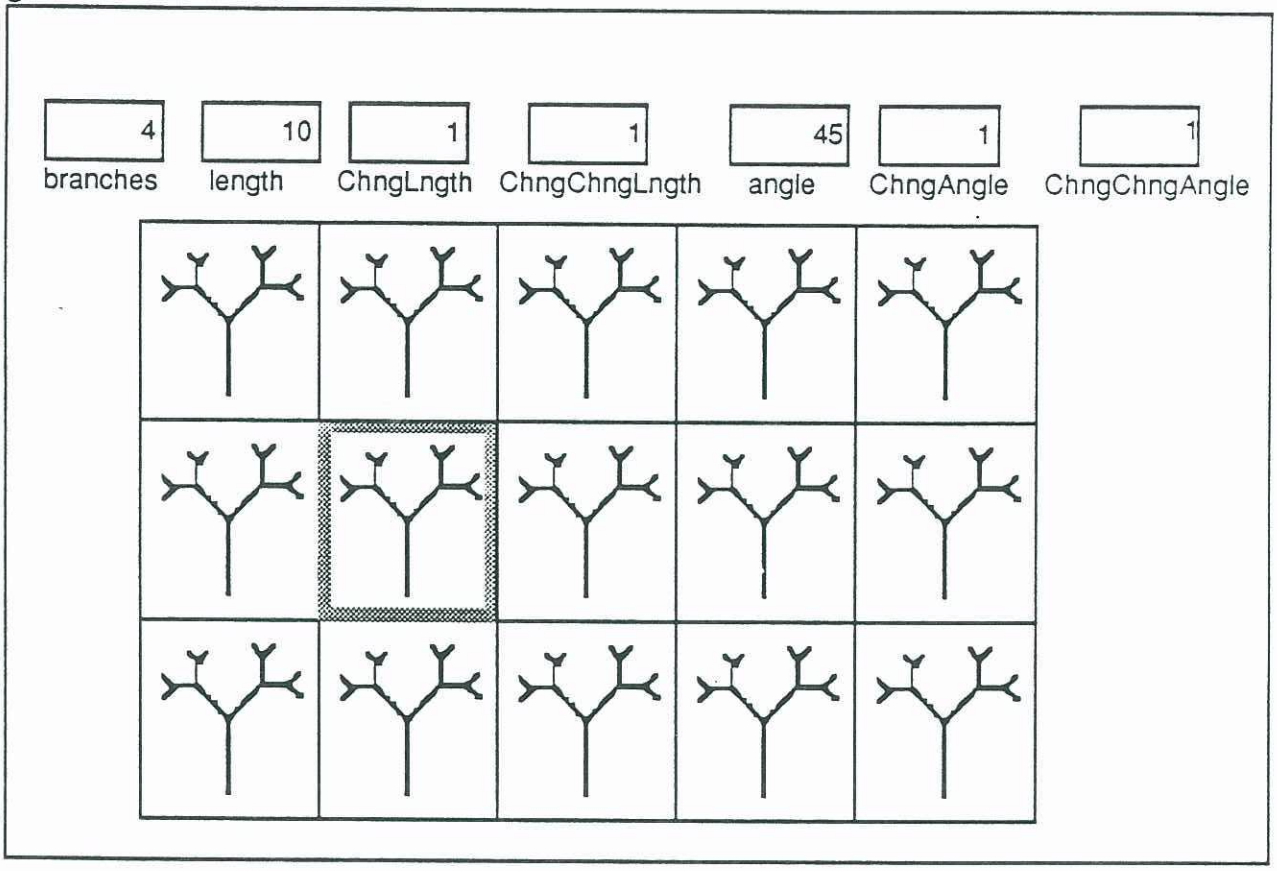


The script is changed to use the additional genes:

```
makeTree =
    tree branches  length ChngLngth  ChngChngLngth  angle  ChngAngle  ChngChngAngle

tree branches length CIngth CCIngth angle Cang CCang =
    if branches = 0 then quit
    forward length                                  -- make a branch
    right angle                                     -- turn to the right
    tree branches-1                                 -- make a smaller tree
        length*CIngth
        CIngth*CCIngth
        CCIngth
        angle*Cang
        Cang*CCang
        CCang
    left 2*angle                                    -- turn to the left
    tree branches-1                                 -- make a smaller tree
        length*CIngth
        CIngth*CCIngth
        CCIngth
        angle*Cang
        Cang*CCang
        CCang
    right angle                                     -- turtle to original angle
    back length                                     -- turtle to original position
end tree
```

"MakeTree" fires off when any of the field-players are changed.

To get Dawkins' variation field is a simple matter of making 14 more tree makers and arranging their views. Notice that we show all of the tree views but only the genes of the selected tree.



The simplest way to do this in Playground is to position all the genes in the same place and then the selected player brings itself to the top:

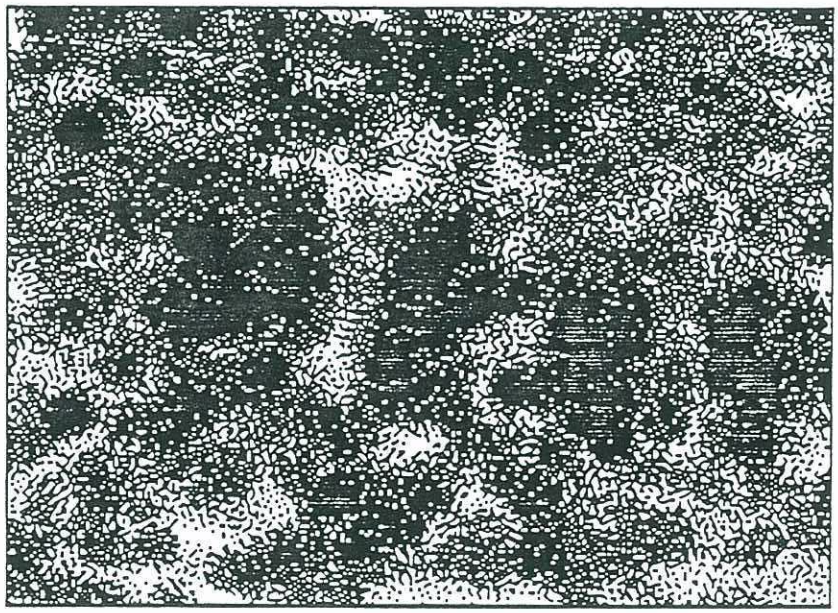
```
Reveal =
    when I'm selected
    bring myself to top
```

An Adventure Game

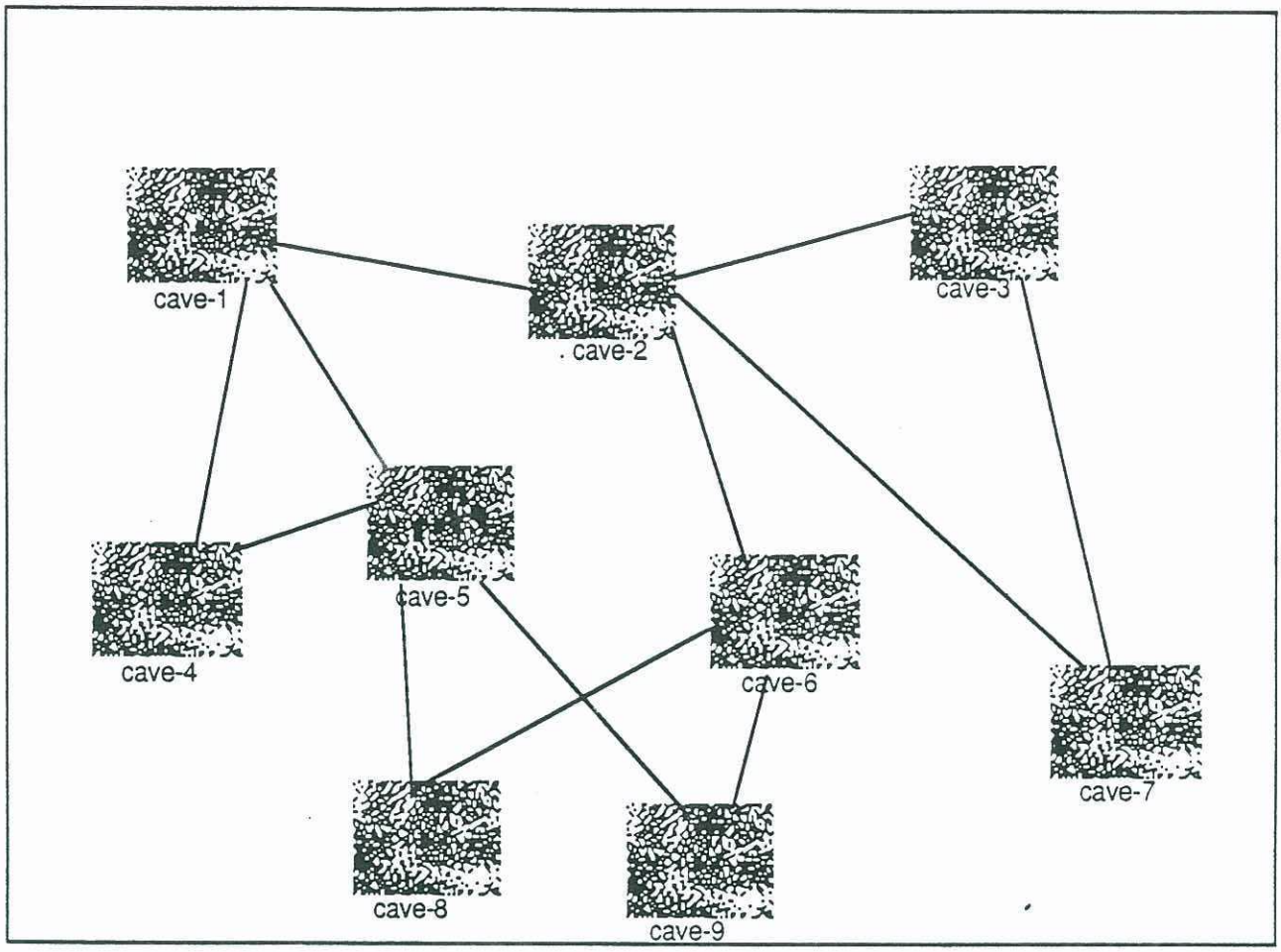
Adventure games have a *quest*, to accomplish a *goal*, *places* in which things happen, *routes* that take you from place to place, and *tools* that can get you by *monsters* and *obstacles*. One of the simplest adventure games is Wumpus, set underground in connected caverns in which the Wumpus slowly roams looking for human flesh. We have warning of the Wumpus's approximate location by a faint smell if he is two caverns away, and a strong smell if he is one cavern away. The most pared down version—in which the Wumpus doesn't move, and the smell is detectable only one cavern away—is a good programming project for a child.

The game is played from the point of view of the intrepid explorer and can be developed and enjoyed from a single view, but it will be very convenient to have a second view that shows all the caverns and makes it easy to come up with new configurations.

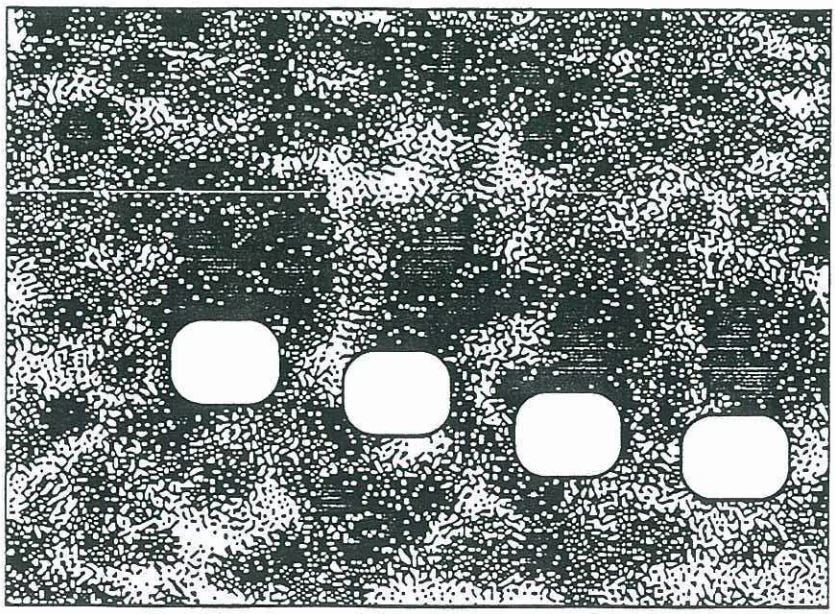
First we draw a typical cavern to use as a background.



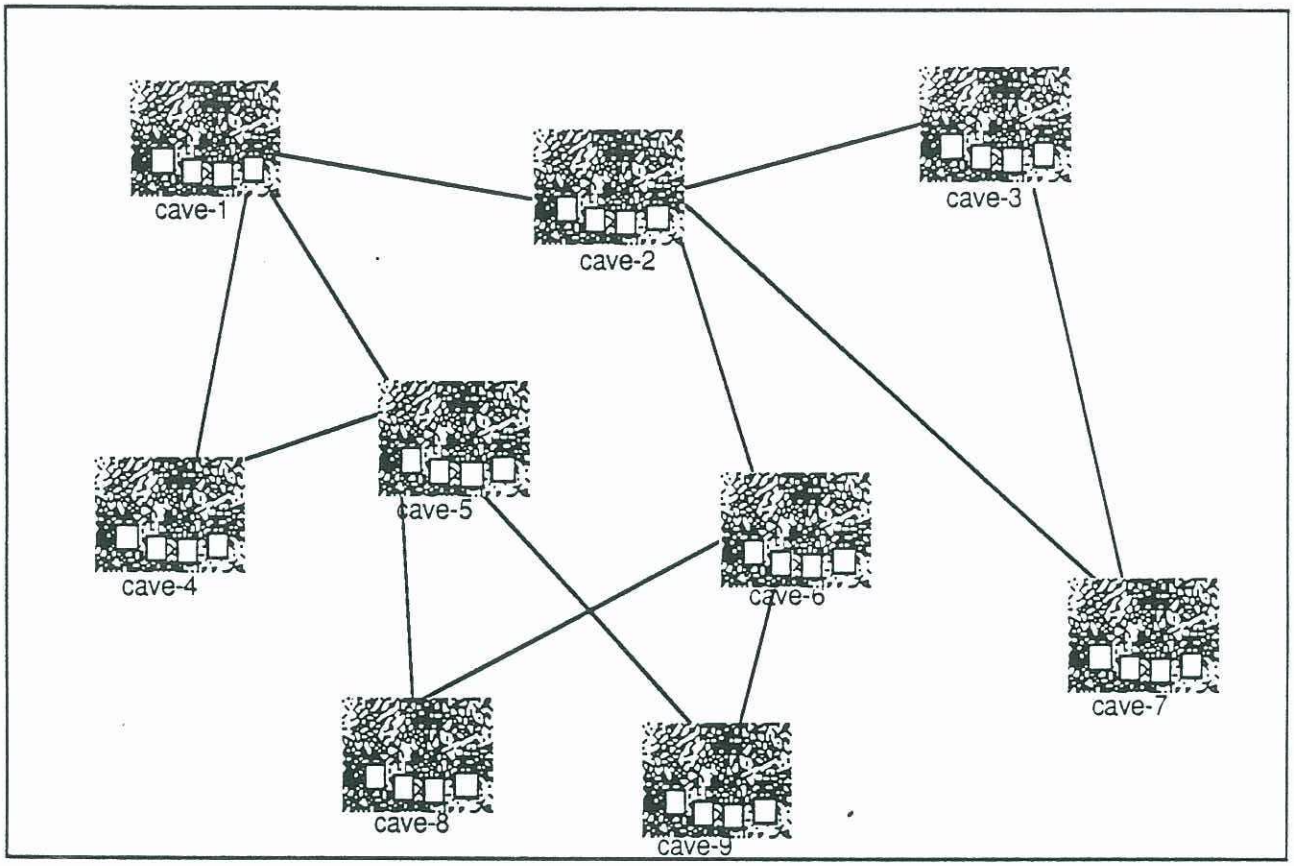
Next, we make a lot of caverns and connect them together with lines to get a sample configuration. The lines don't have any meaning yet—they are just there to help plan. Later we will get the view to generate them automatically from the connections.



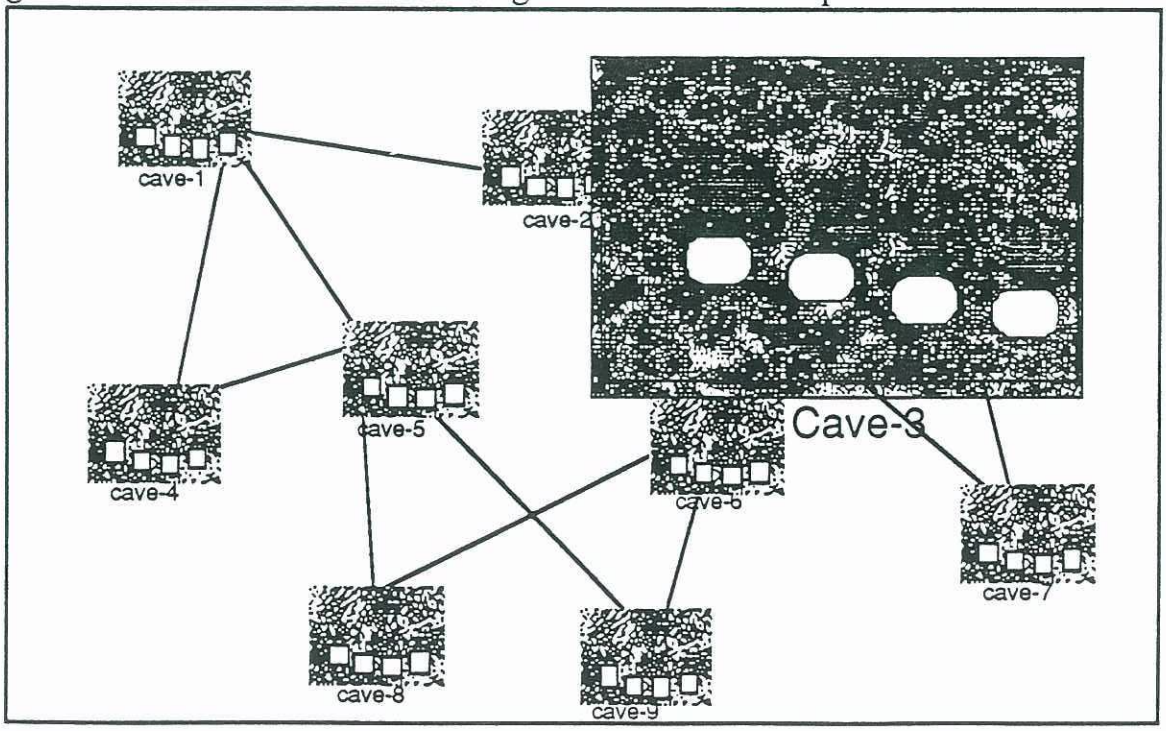
Now we need a way to hook up one cavern with another that lets us make easy changes. We decide to use player-fields that hold the number of the cavern they connect with. We put them into the background.



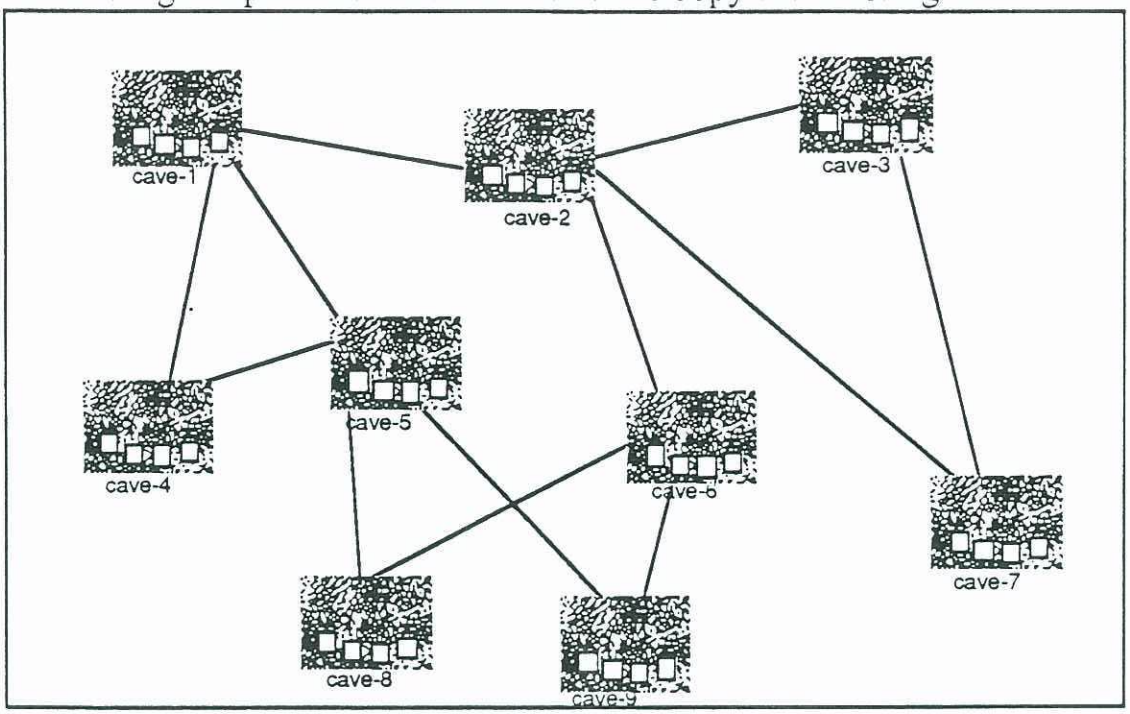
All the caves now show the fields.

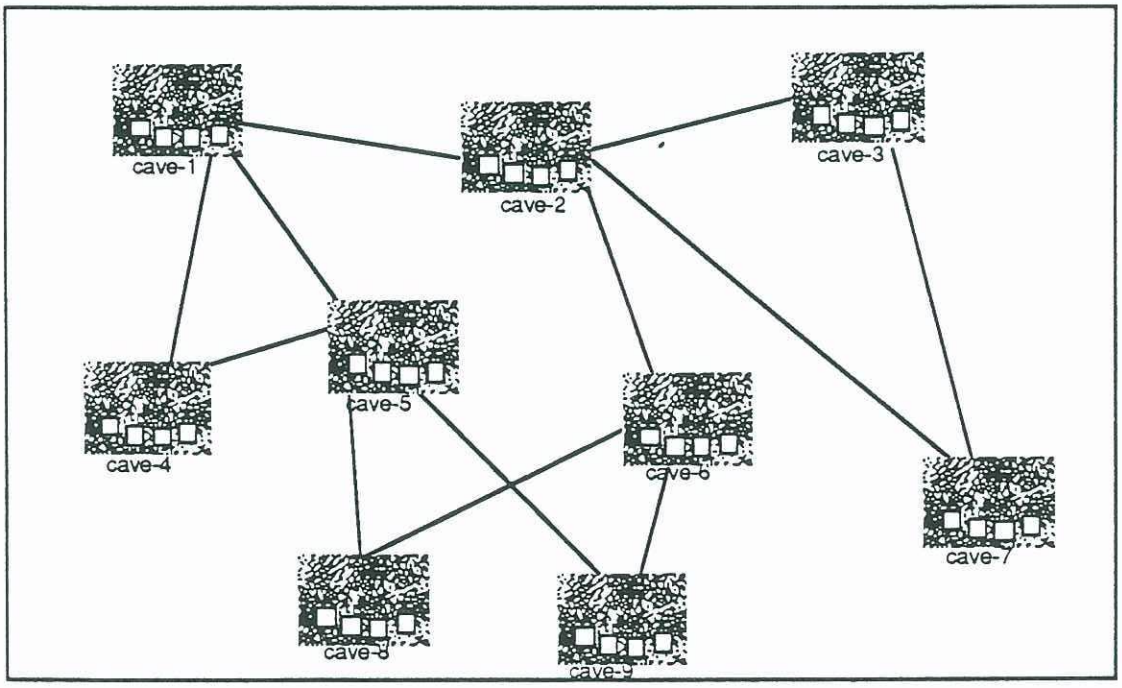


Now all we have to do is to connect them. It would be convenient if they were larger. But when we make them larger we can't see the planned connections.



This is a good place for another view. We copy the existing view.

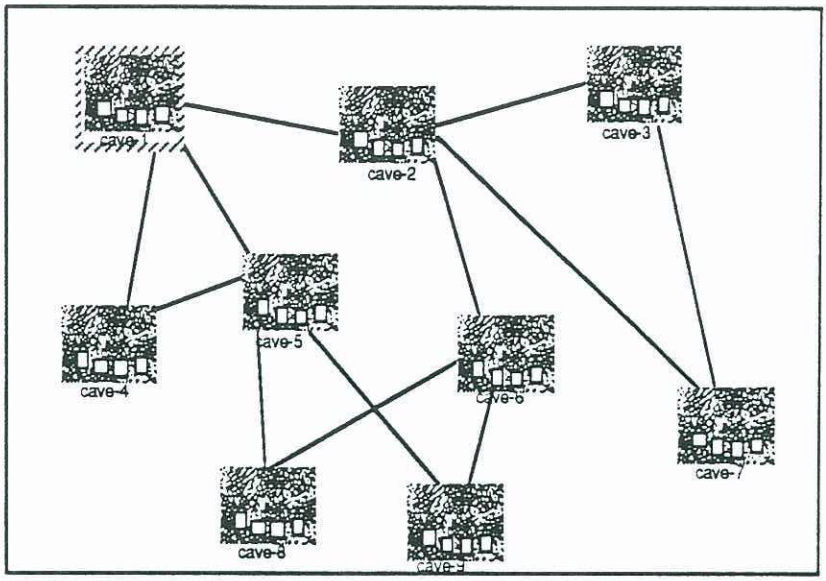


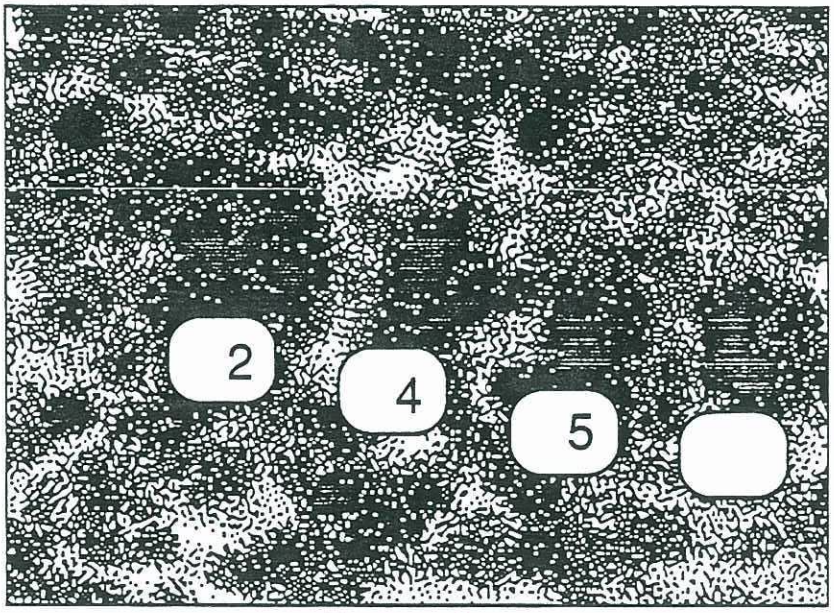


Now we add a script that will make anything selected in view-1 be brought to the front and magnified in view-2.

```
cave's view-2 magnify =
        when view-1 selection is a cave
        get id of view-1 selection
        set its layer to "front"
        set its extent to my extent
```

Now it is easy to fill in the connections—and we have motivated how the game will be played.





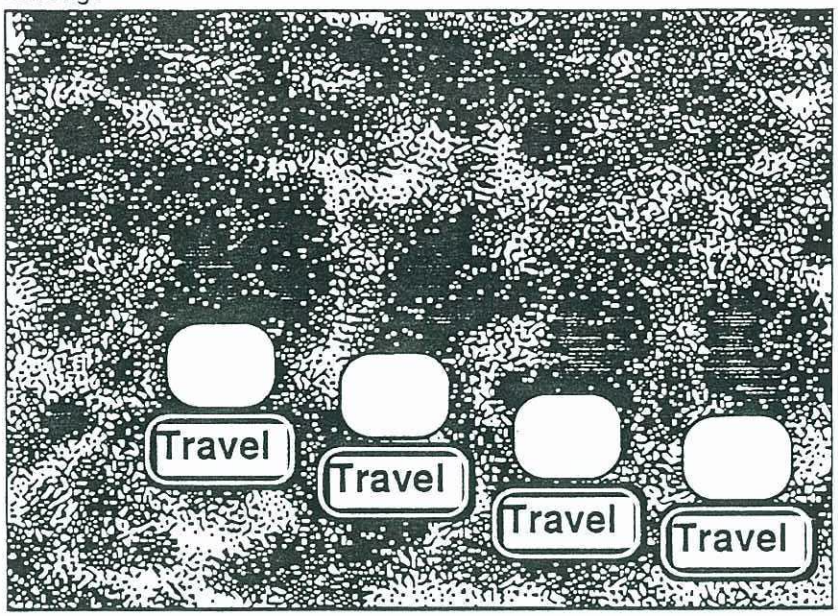
Cave-1

Having connected the caves together, we now have to think about how we move around the caverns. The easiest way is just to get another bunch of players to act as travel buttons and place them below their corresponding passageways. We can do this in the background since we want each button to do the same thing.

The script is written spreadsheet style.

```
cave's travel =
        when mouseUp
        get id of <here we point to the corresponding field to get ...> player-1
```

```
In view-1 select cave number of it's first word
put it into message
```



Background

We replicate the buttons spreadsheet style so that the relative correspondance is maintained. Each button causes a selection in view-1 which in turn causes a blown up version to appeár in view-2. Now we are traveling around the caverns.

Now for the Wumpus. We make another player to be the beast—all it needs for now is the label "Wumpus". We make a player-button called "start" whose job it is to place the Wumpus in a randomly selected cave.

```
cave's start =
    when mouseUp or receipt
    get random (number of caves)
    move Wumpus to cave-it
    in view-1 select cave-1
```

Now it remains to furnish the game-player with the smell clue. We go back to the background and put in a new player-field called "air" that continuously monitors the neigboring caverns for the Wumpus and transmits the odor back.

```
cave's air =
    when player1 or player 2 or player 3 or player 4 contains a player called "Wumpus"
        then view = "A Disgusting Smell"
        else view = "No Smell"
```

This particular piece of code probably shouldn't and couldn't be written this way. (To see the way I did this in HyperCard with a single view, see Appendix). A better

(more object-oriented) way to do it might be to have the field-players be standins for the cards whose indices they hold. Then the above code would work, and some of the previous code would be simpler.

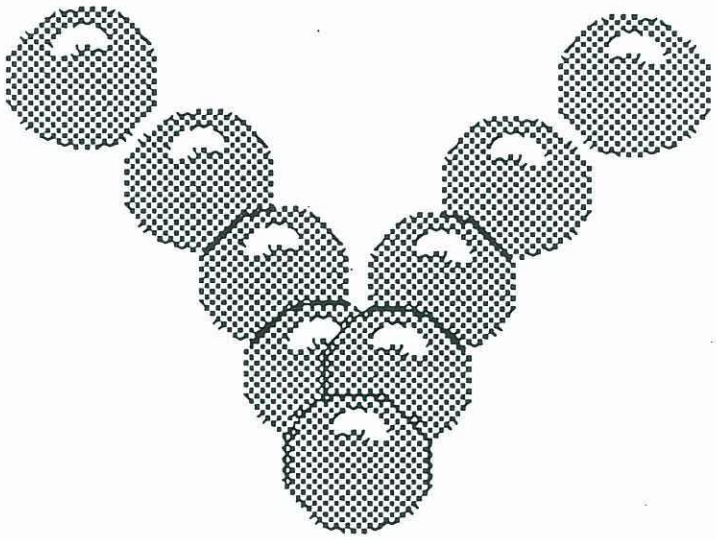
In the model part we will have a collection of players, each one will be a cavern, and we only want to see one at a time. The model is like a hypercard stack and we need to get the view to show us just one "card" at a time. An easy solution is to have each player's view-script react to a selection by bringing the selectee to the front and blowing it up so its view is the same size as the viewing window. This amounts to defining "go to" but without the pernicious side effects of changing the context of code that hypercard has. A more useful way to accomplish the same thing would be to start with all the caverns to the side of the view, then to move just the selected one to the center blown up. That way, the repetoire of caverns is always visible as small images and provides a motivation for the two view version. We also have to program "next" and "previous" so we can cycle through the caverns. All these are so useful that they are probably already built in, but it is instructive how easy they are to do from scratch.

Traditional Examples

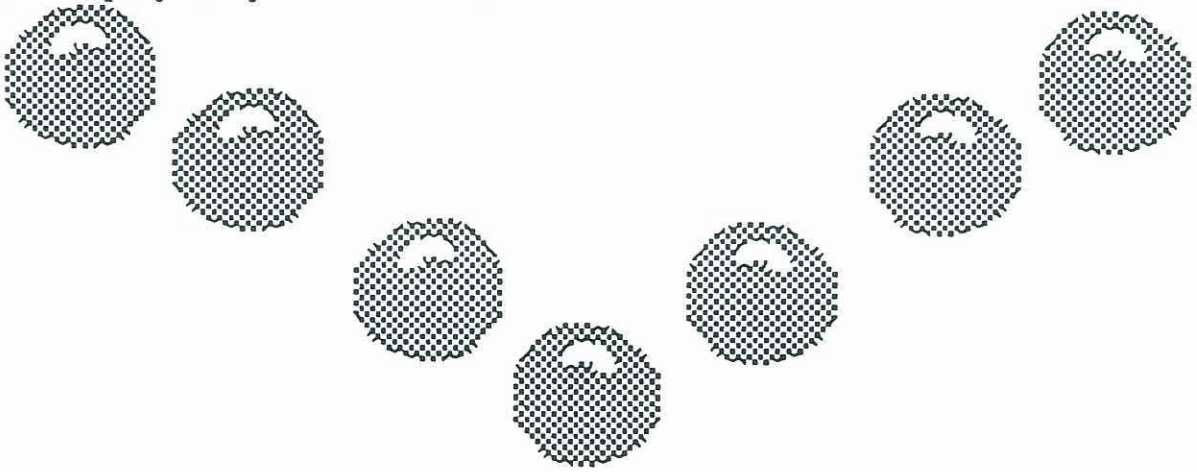
Regardless of the theoretical niceties to come, the following have to work as simply and straightforwardly as possible. Here we grapple with the "complexity" need to have goals be modular players vs. the "user interface" need to have extended scripts.

Bouncing Ball

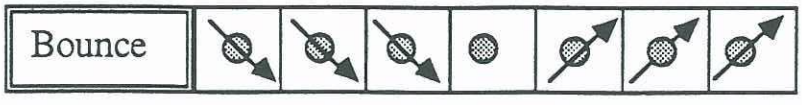
This is the most venerable of traditional animations. Grab the brush or the circle tool and make a ball. Choose **recordStory** from the ball's menu. Move the ball with the mouse in a bouncing path.



Choose **playStory** from the ball's menu to watch it bounce.

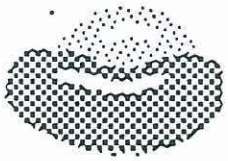


Look at the story to see what has been recorded.

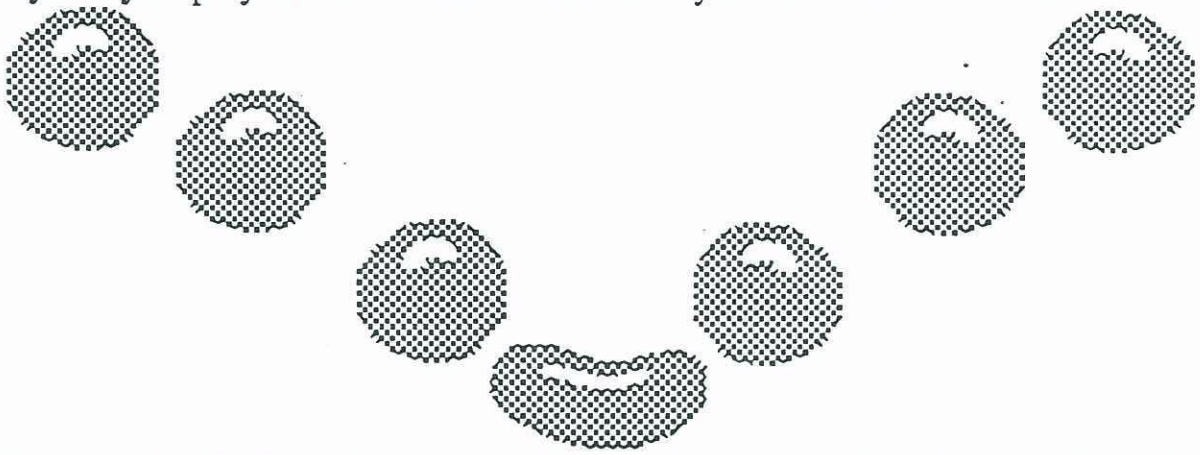


It is a sequence of movements (and hence positions) of the ball to be attained. Each story in Playground is always a sequence of goals. At this level the players look like a storyboard. When expanded, more detail can be seen.

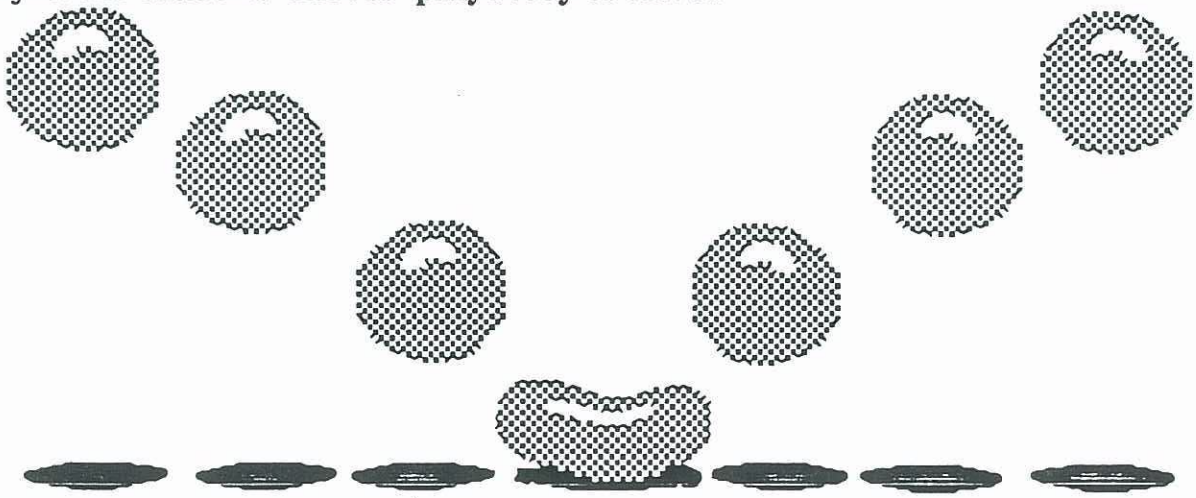
Choose **stepStory** to step to the lowest point. Choose **newView** to make a flattened ball. Note that the old view stays behind to help register the new view.



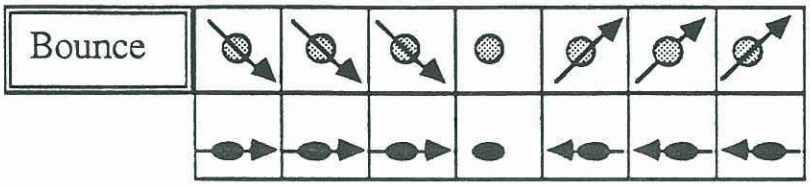
Delete the old view from this frame when done. Choose **rewindStory** and then **playStory** to play animation which now nicely deforms when it hits.



To put in the shadow, choose **track2** and **step** to position along both tracks as you adjust the shadow. Choose **playStory** to test it.



Now choose **renameStory** as **Bounce** to save the bouncing behavior as a goal that can be accomplished by the ball.

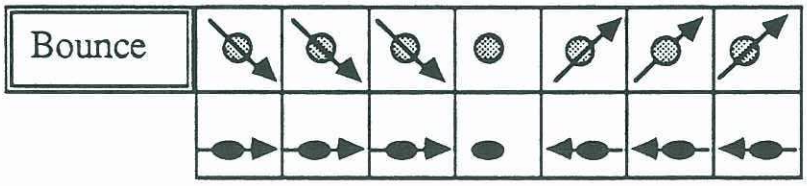


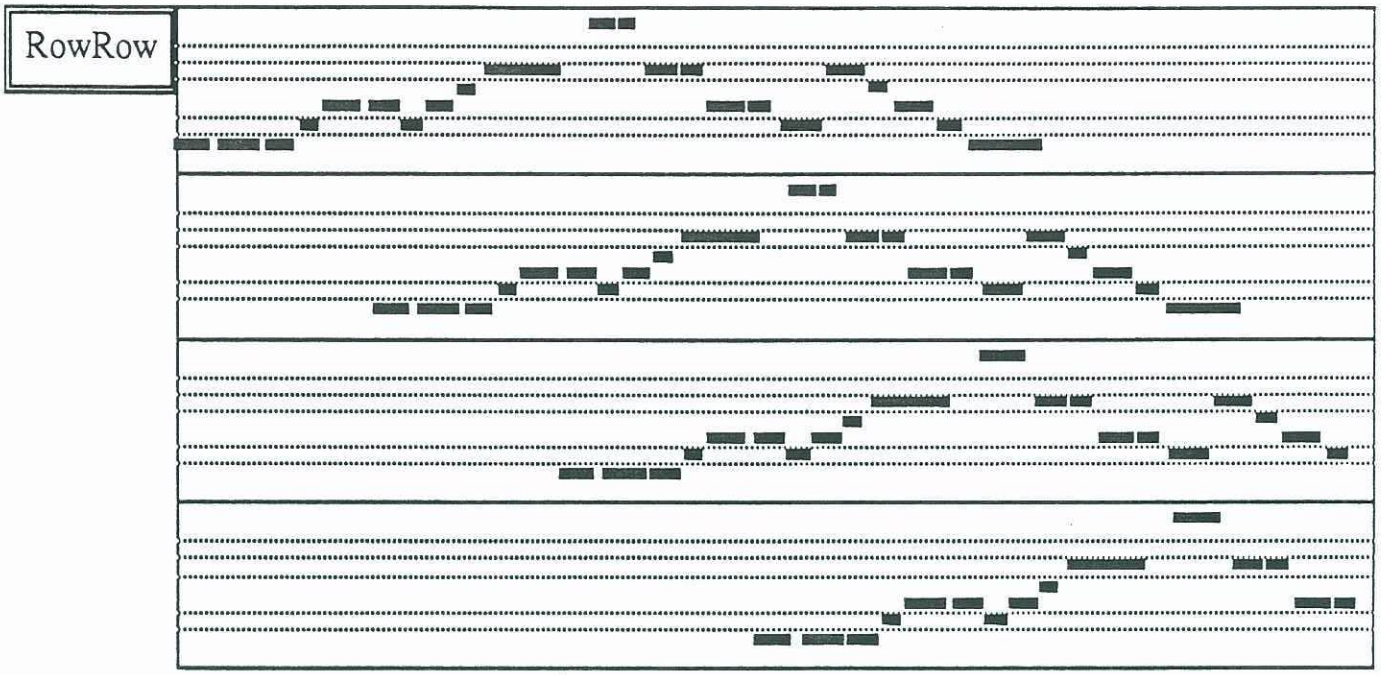
It is critical that user interface actions get turned into individual player-goals (containing sequential goals) all of which are embedded in the actual "actor".

Row, Row, Row Your Boat

Works exactly the same as visual animation except animates sound. Use the keyboard or hummer tool to make a player. Choose **recordStory** from the player's menu. Touch the notes with the mouse to play the tune. Choose **playStory** from the player's menu to hear it play. Change the length of any note by grabbing its handle. Change the pitch by moving the note vertically. To put in the other voices, choose **track2** and **step** until you hear where the next voice should come in. **Copy** the whole tune in **track1** and paste it into **track2** at the new starting point. Choose **track3** and repeat for new voice entry, and then do the same with **track4**. Finally choose **renameStory** as **RowRow** to save the round as a goal that can be accomplished by the player.

It is worthwhile to examine the stories in more detail.





Etc... they are completely alike except for the last instant in which one gets turned only into an animated image and the other into sound. The latter implies that we should be able to render the ball's animation as sound also. Get a new instrument from the ToyBox. Open its **story**. Copy the ball's animation script—just as you would text in a document—and paste it into the new instrument's **story**. Now **playStory** and hear what it sounds like! Again, as with the single frame deformation of the ball, we have an opportunity to insert a special effect noise in the frame when the ball hits the ground. Notice that the view of the animation story changed to look like notes. This means that "paths of relative movement" have many different views and can be used in many different contexts. Can you think of some more? How about:

Each of the animations got moved from event to event by noticing the local clock built into the story controller. Their attention can also be directed to take cues from outside. For example, let's have the ball pay attention to RowRow. First, let's have the ball move a frame on each note of RowRow—this will be jerky but surprisingly effective. Now, let's have the ball notice RowRow's basic beat. Conversely, we can try to get the ball to hit the ground on each note event of RowRow. The easiest way to

do this is to get RowRow to pay attention to the ball and to play a note every time the ball hits. This works but destroys the rhythm of the tune. The more complicated way that works is to have the ball ask what RowRow is planning to do next and then plan its own Bounce tempo accordingly.

An interesting experiment is to take the four tracks of RowRow and move them over each other to see if we can understand how a round works. Notice that each vertical time segment made up a chord—now it is easy to see. Play each one separately to test. So one way to compose a round is to take a nice chord sequence and then pick out a melody from each of the voices of the chords untils all have been used. Let's try it.

<<etc.>>

A canon is a round in which we change key for each voice entrance:

<<etc.>>

To finish the analogy between animation and music, we turn the ball into a frog (ala videoworks) and animate multiple tracks to bring on "the plague of biblical proportions".

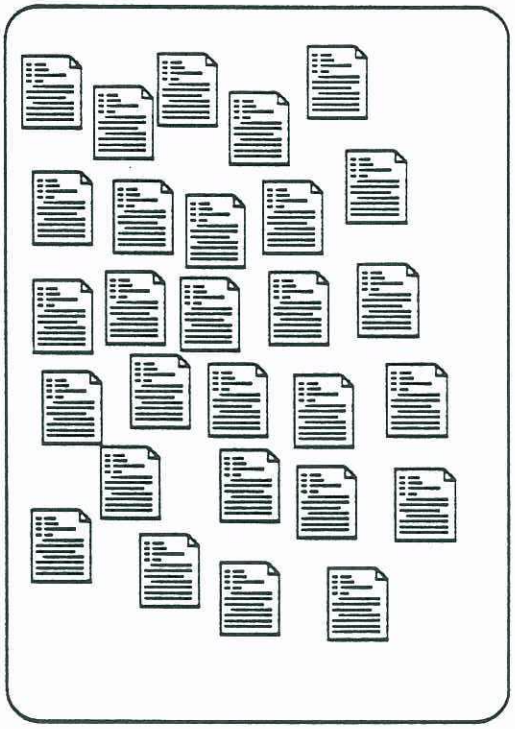
For indeed, the "plague" is nothing more than a simple canon in which changes of "pitch" of the basic theme are represented as changes in the y axis of the starting position (just as they are in standard musical notation).

Electronic Mail Server

Playground's commons area—in which anything that happens there happens in every commons on the network simultaneously—provides an opportunity for an end-user to build a very nice electronic mail server.

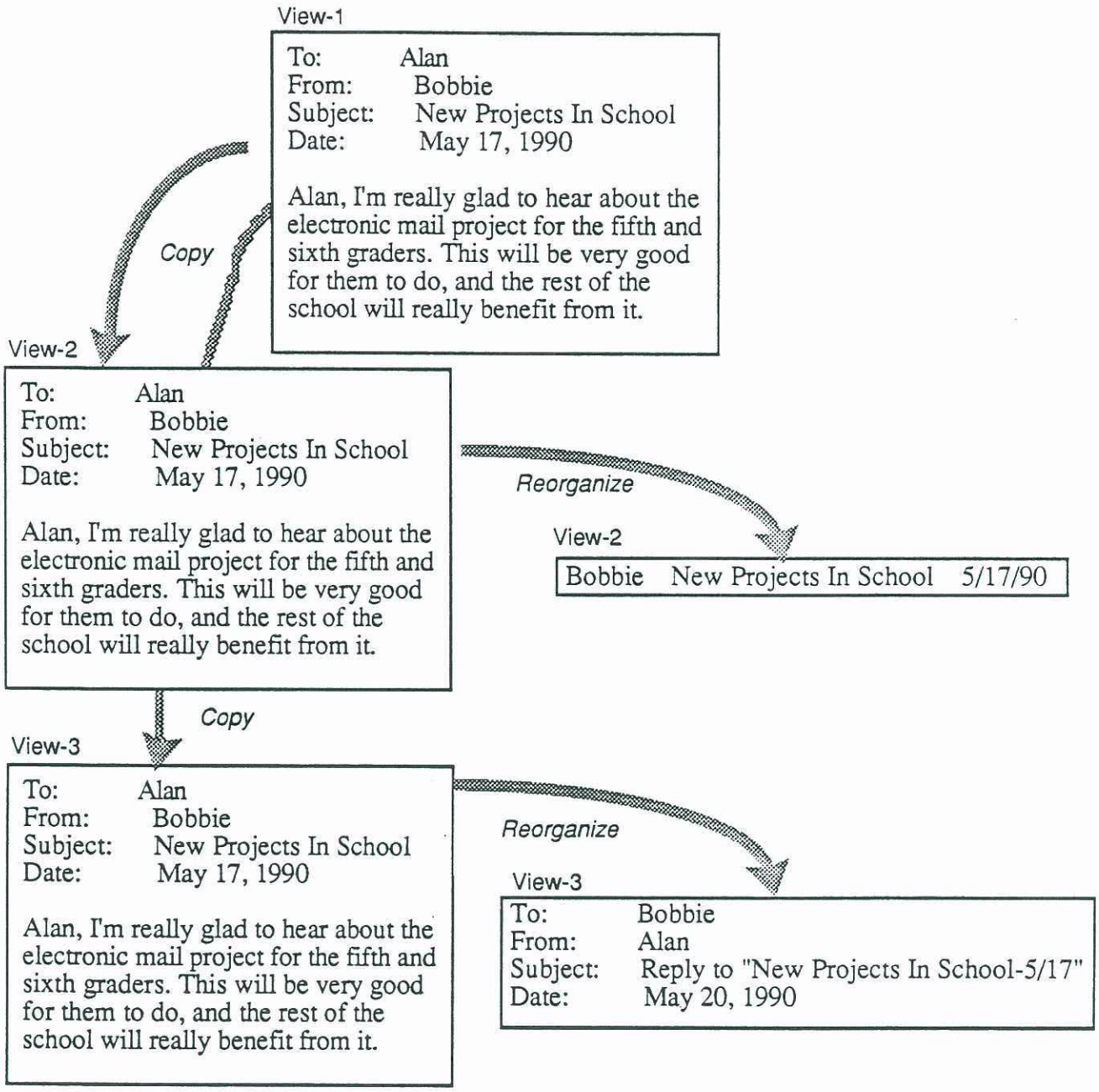
```
MailBox =
    addto value the fetch of find "alan" in field "To:" of mailBag in Commons
```

MailBox

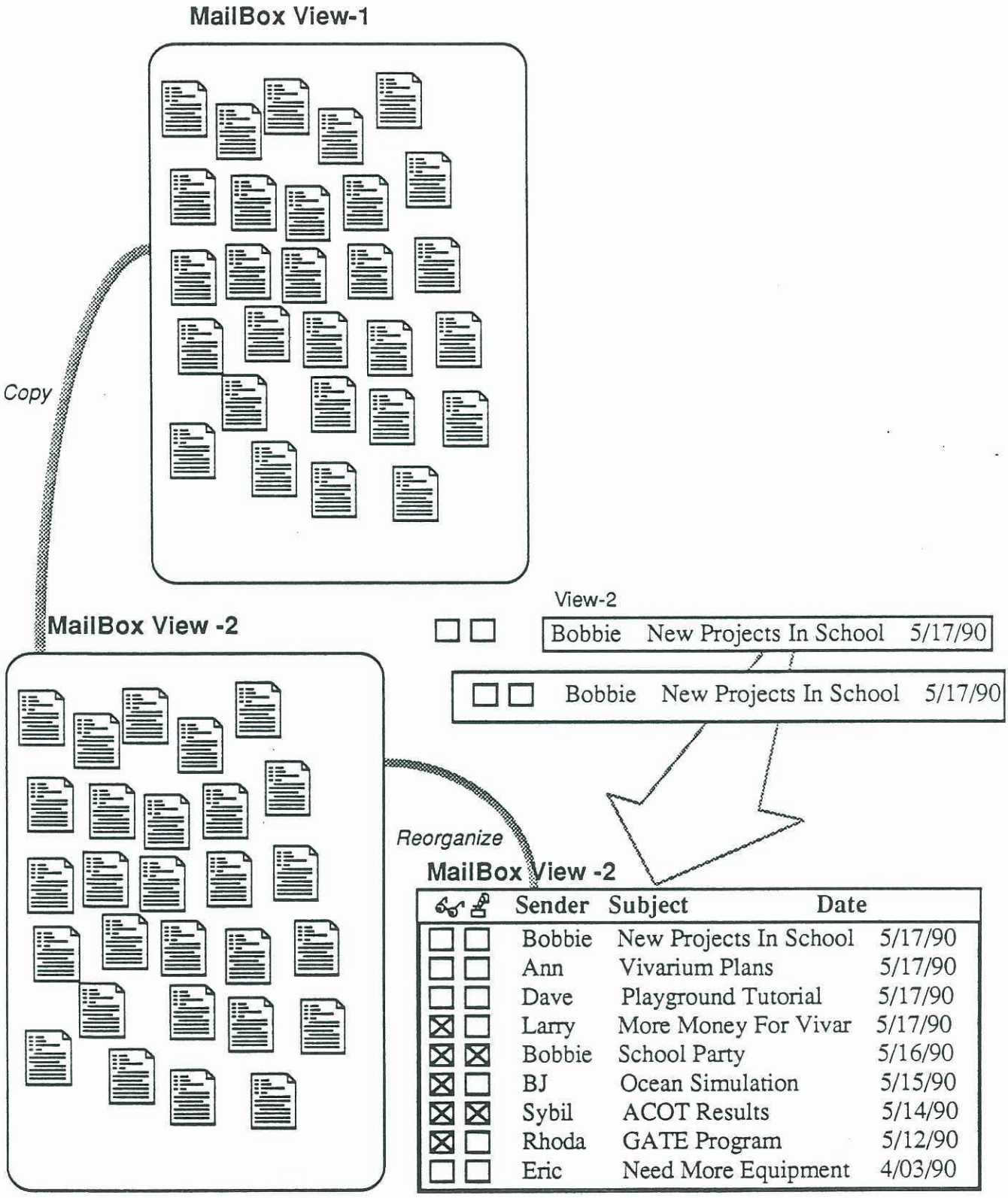


Later we can think of more intelligent ways to ferret out useful information in the commons—perhaps even adapt one of our fish models to think that stuff we are interested in is food or nesting material and should be constantly searched for and brought back. For now, this will do. Everytime a player that has my name in the player-field "To:" is put in the the commons' mailBag, it will be moved to my personal mail box on my machine.

To make a mail server, we need to make some new views. First, a couple of views at the document level. We open the icon of a mailgram to get its main view and copy it twice to get two new views for a mailgram. The first of these is organized into a summary line that shows the sender, the subject, and the date. The second copy is organized to show the reply heading.



Now we can build the actual views for the mail server. It is modeled after the legendary "Laurel" server at Xerox PARC. First we build the summary view that lets the user know what mail is in the mail box and what has been done with it. We copy the mailbox view and modify it.



A line-view in the mail box summary view is constructed by taking the mailgram summary view we just built and combining it with two button-players, one to read the mailgram (and then to show that it has been read), and the other to show that a

reply was actually sent.

To get the read view we copy the mailbox view again and give it a script that will react to a selection in the summary view.

```
MailBox View-2 Magnify=
    When Summary-View selection
    Get Model of Summary-View selection
    Set its layer to "front"
    Set its View to View-1
    Set its extent to my extent
```

This will find the underlying mailgram whose view-2 is selected in the summary line, move it to the front, select its main ("Open") view, and size it to the read view's rectangle.

```
Read Button =
    When Summary-View selection is my owner
    set value to "X"
```

Once read, the button is always marked. We now have constructed two thirds of the mail server. It remains to cause the reply mailgram to be generated whenever a read is done.

MailBox Server

	Sender	Subject	Date
☐ ☐	Bobbie	New Projects In School	5/17/90
☐ ☐	Ann	Vivarium Plans	5/17/90
☐ ☐	Dave	Playground Tutorial	5/17/90
☒ ☐	Larry	More Money For Vivar	5/17/90
☒ ☒	Bobbie	School Party	5/16/90
☒ ☐	BJ	Ocean Simulation	5/15/90
☒ ☒	Sybil	ACOT Results	5/14/90
☒ ☐	Rhoda	GATE Program	5/12/90
☐ ☐	Eric	Need More Equipment	4/03/90

To: Alan
From: Bobbie
Subject: New Projects In School
Date: May 17, 1990

Alan, I'm really glad to hear about the electronic mail project for the fifth and sixth graders. This will be very good for them to do, and the rest of the school will really benefit from it.

To: Bobbie
From: Alan
Subject: Reply to "New Projects In School-5/17"
Date: May 20, 1990

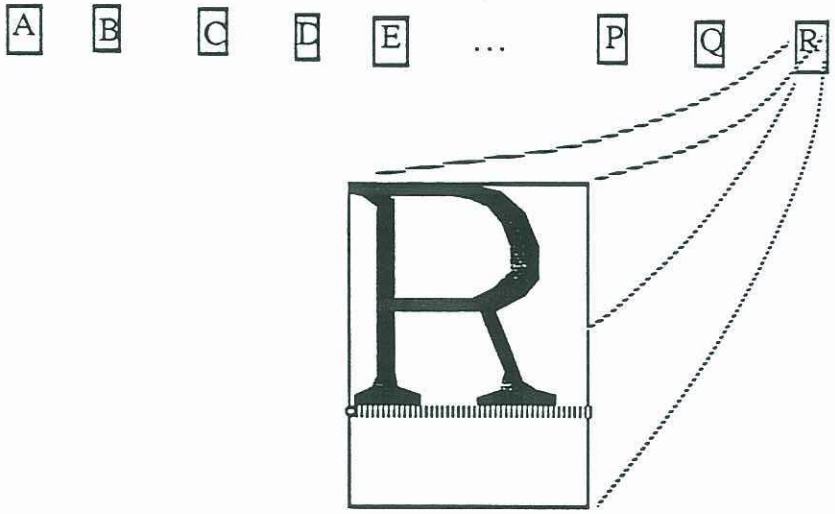
We make a "send-mail" player-button whose action is to put the reply back in the commons' mail bag. The "have sent" button in the summary view is noticing what this button does and changes its state when the mail is sent.

All the views should be really be scrollable views, etc. There are other details that have been omitted in order to treat this with a broad brush.

A Simple Word Processor

We want to construct a MacWrite-style paragraph editor. First we gather the novice user's guesses about the components of a paragraph: that there are individual *characters* strung together, any subrange of which can be selected—in addition, there are *words* and *lines* that can be selected. There is a *selection*. A single click selects a zero-width *gap*. A draw-through selects a *range* of characters. A double click selects a word. A shift-click extends either the character selection or word selection. Any typing of non-control characters *replaces* the selection. The delete character replaces the character before the zero-width gap with nothing. If the selection is not zero width, it replaces the characters in the selection with nothing. There are control characters for changes of font, highlighting, and emphasis.

We start with a limited Playground toolbox. The generic object is a rectangle that can own objects (with properties similar to OrderedCollection in Smalltalk).



First we make a font (say "FakeTimes") by drawing each character, then place them in a generic object to hold them in order. We hook the keyboard up to the font with the rules:

```
currentFont = FakeTimes.
currentKey = if keyboardAction then currentFont at: key.
```

We have two main tasks in the editor itself. First, the display of the characters in lines using proper breaks on word boundaries when a line margin is exceeded. And second, the selection of characters using the mouse and replacement of the selection with new typing.

The characters will be held in order in a generic object called Chars. We set up

another generic object, Words, to hold the words formed by the characters. Now, how to find the words? If the chars were children lined up and connected with rubber bands, they would solve the problem as follows by coming up with four simple rules.

Chars need to find what word they are in. They do it by seeing if they are the same kind of character (space or nonspace) as their predecessor.

```
CHARACTER
my word =       if I'm first then word 1
                if I'm a space then the word after my before's word
                if my before is a space then the word after my before's word
                if my before is not a space and I'm not a space
                    then my before's word.
```

Words need to find what line they are in and where they are placed. They do it by seeing if their total width will allow them to fit into the line so far.

```
WORD
my chars =      Chars whose word = me.
my width =      my chars' width summed.
my rtMargin =   my basePoint's h + my width.

my line =       if I'm first then line 1
                if my width + my before's rtMargin ≤ my before's line's rtMargin
                    then my before's line
                if otherwise then the line after my before's line.

my basePoint =  if my line = my before's line
                    then my before's basePoint + (my before's width @ 0)
                if otherwise then my line's basePoint.
```

A simple rule for the postion and extent of lines is to calculate it from their owner's rectangle and position in the line list.

```
LINE
my basePoint =  my owner's h @ lines loc: me * fontHeight + leading.
```

```
my rtMargin =      my owner's extent h.
```

In order to display and for selections to be noticed, the individual characters have to know where they are. So we go back and add to each character the rules that enable them to figure out where they are from the word they have chosen.

```
CHARACTER
my basePoint =     if I'm first in my word then my word's basePoint
                   if otherwise
                        then my before's basePoint + ( my before's width @ 0).
```

We have finished the first task. All the chars will display themselves in the proper position. They are also ready to notice the cursor as we move on to the second task: selecting and editing.

A selection is a temporary collection. For this example we will only use (as does MacWrite) a contiguous subcollection of characters. Our purpose is to find Chars **from**: i **to**: j, not just collect the individual characters, because the main editing operation (replacement) affects Chars itself. The other operations of font and size changing can be distributed to the individual characters in either case.

```
SELECTION
my startPos =      if mouseDown then Chars whoSeesMe - 1.
my curPos =        if mouseDrag then Chars whoSeesMe - 1.
highLight =        Chars from: 0 to: startPos noHighlight.
                   Chars from: startPos to: curPos highlight.
                   Chars from: curPos to: end noHighlight.
edit =             if key ≠ CTRL
                        then Chars from: startPos to:curPos = currentKey
```

We could also have split it up differently:

```
SELECTION
my startPos =      if mouseDown then Chars whoSeesMe - 1.
my curPos =        if mouseDrag then Chars whoSeesMe - 1.
```

```
CHARS
highlight =     if selection is trying curPos
                    then me from: 0 to: selection's startPos noHighlight.
                        me from: selection's startPos to: selection's curPos highlight.
                        me from: selection's curPos to: my end noHighlight.
edit =          if key ≠ CTRL
                    then me from: selection's startPos to: selection's curPos = currentKey.
```

Or even have given it all to Chars and not have a separate Selection. In this simple case it really doesn't matter, though perhaps the second example is more elegant because we have cooperative noticing and tracking between the Chars and Selection objects. For the general case in Playground, we will want to have a general Selection collection that can gather up a set of collection references (via shift-click for example)—which will in turn listen to the keyboard or the PASTE command and then properly distribute the replacement objects. A generalized UNDO is much easier this way also.