Smith, David C., and Cypher, Allen. "KidSim: Child Constructible Simulations". In Proceedings of the Imagina '95 Conference, Monte-Carlo, February 1995, pp. 87-99.

KidSim: Child Constructible Simulations

David Canfield Smith, Allen Cypher
Apple Computer, Inc.

Introduction

Simulations are a powerful tool for education. They encourage unstructured exploratory learning. Simulations allow children to *construct* things, helping them attain a deeper understanding than is possible from books alone. As the poet and writer Cesare Pavese says: "To know the world, one must construct it." And there has never been a better tool for running simulations than the computer. Computers can handle far more complexity than people can with pencil and paper. Children can formulate hypotheses and conduct experiments to test them. The beauty of simulations is that kids' hypotheses can be *refuted*. It is impossible for a child to remain passive when this happens.

Although most simulations today allow users to modify some of their parameters, they do not permit users to modify their fundamental behaviors and assumptions. These are built in and can be changed only by professional programmers. This inflexibility makes them too rigid to support many teachers' curriculum goals, and so many school teachers do not use them in the classroom. Simulations which do allow fundamental modification invariably require extensive programming skills. Few children or teachers can or want to do it.

What's needed is a way for children without programming knowledge to have complete control over the behavior of simulations. What's also needed is a way for teachers to tailor simulations to advance their curriculum goals. KidSim provides a way to do both. In KidSim, children define simulation behavior by programming the objects in the simulations, but they do so in a new way, what we call "languageless programming." In essence our approach is to apply the good user interface principles developed during the 1980's for personal computers to the *process of programming*. The key innovation is to combine two powerful techniques: graphical rewrite rules and programming by demonstration. Programming by demonstration builds programs in the background while children manipulate simulation objects as they would normally. Graphical rewrite rules provide a representation for programs that is understandable to children. The combination appears to provide a major improvement in kids' ability to program simulations.

The Elements of KidSim

KidSim is an environment for building extensible simulations. It is fully described in [8] and [2]; here we just give a brief overview of its elements. An "extensible simulation" is a computer-controlled microworld whose behavior can be modified by users at run time. KidSim provides the following tools for building such simulations:

- The *game board* represents the simulation microworld. It contains various "simulation objects" which, as time progresses, move around on the board interacting with each other. It is divided into discrete spaces, like a checkerboard.
- The *simulation clock* starts and stops the simulation running and can run the simulation backward, giving KidSim a multiple undo capability. Time is divided into discrete ticks.

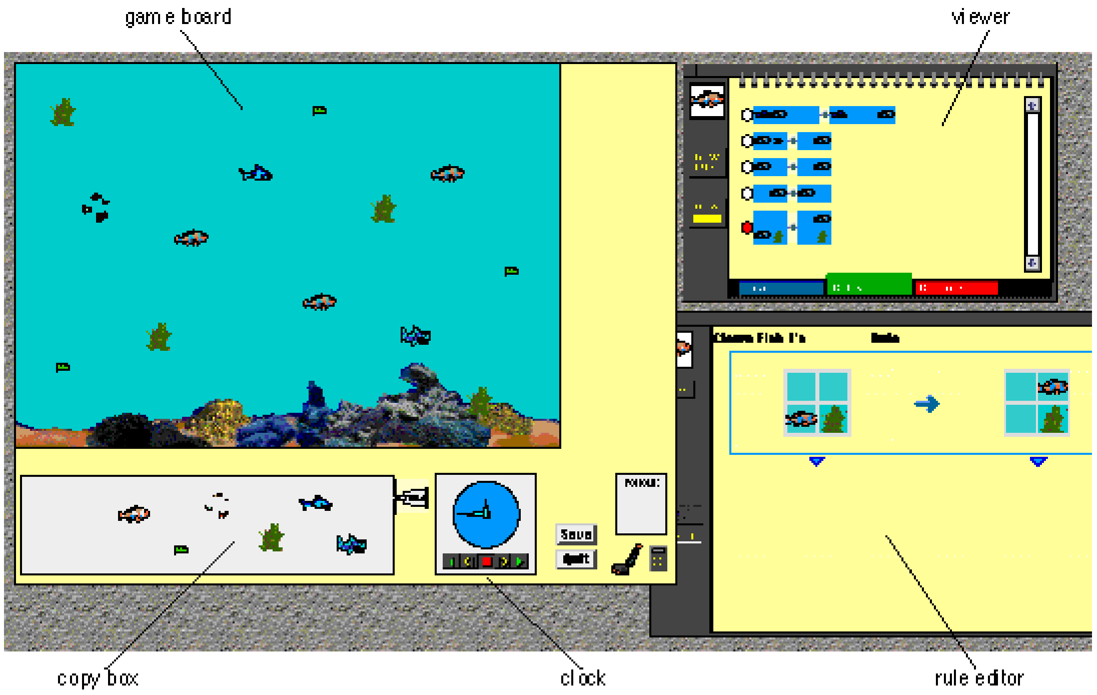


Figure 1 - a KidSim ocean simulation

* The *copy box* is a source for new simulation objects. Each time an object is dragged out of the copy box, KidSim makes a copy of it leaving the original unchanged. Thus the copy box has an infinite supply of objects.

* Each simulation object has a *viewer* which displays the rules for its behavior, its symbolic data in the form of name-value pairs, and buttons that allow it to be manually controlled.

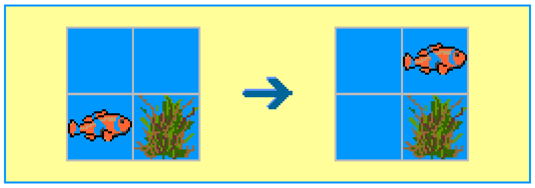
* The *rule editor* is the environment in which users define and modify rules to control an object's behavior.

Programming in KidSim

KidSim takes a new approach to programming by eliminating the programming language. Instead, we are applying good user interface principles, the same principles that guide nearly all personal computer editors today ([1], [9]), to the *process of programming*. KidSim programmers *edit* rather than *program* in the traditional sense. Instead of using a text editor to write language statements which abstractly describe actions that occur later, KidSim programmers create graphical rewrite rules by directly manipulating simulation objects. Editing is easier for most people than programming.

Graphical rewrite rules

A *graphical rewrite rule* is a transformation of an area of the game board from one state to another. It consists of two parts: a left side and a right side. Each side is a small scene that might occur during the running of the simulation:



When creating a rule, the left and right sides of a rule are initially the same; all rules start out as identity transformations. Users define the semantics of a rule by *directly editing its right side*. KidSim records the editing actions in a form that can be reexecuted later. This is "programming by demonstration" (see below).

When running a simulation, a rule is said to *match* if its left side conforms to some region of the game board. When a rule matches, KidSim executes its recorded actions, its "program." The effect is that the region of the board which matched the left side is replaced with the scene in the right side.

Graphical rewrite rules are a form of "if-then" rules, which are at the heart of "production systems." There is a long history of production systems in Artificial Intelligence ([4], [6], [7]), mainly as the control structure for expert systems. However, expert system rules are not graphical and have proven to be difficult for nonprogrammers. Typically expert systems require an intermediary--a programmer--to sit between the computer and the user, translating what the user says. The goal of KidSim is to get rid of this intermediary.

Programming by demonstration

Programming by demonstration is a technique in which the user puts a system into "record mode," then operates the system in the ordinary way, and the system records the user's actions in an executable program. The key characteristic is that *the user interacts with the system just as if recording were not happening*. Users don't have to do anything differently nor do they have to learn anything additional. This has proven to be easy for most people who have tried it.

There have been a number of programming by demonstration (PBD) systems prior to KidSim. The major ones are listed in [3]. A problem for PBD systems is how to represent recorded programs in a way that users can understand. Often PBD systems show programs as scripts in some language syntax. But it doesn't work to let people define programs in a way they can handle--by demonstration--and then force them to learn a programming language to examine them.

The main KidSim innovation is *combining programming by demonstration with graphical rewrite rules*. When combined, each solves a problem in the other:

- Graphical rewrite rules solve the problem of how to represent recorded programs. Graphical rules serve as *visual reminders* for programs; they are not full representations. But children don't want to see full representations; full representations are too complex. Rather, graphical rules serve as *hints* as to what the programs do. We have found that children can go down long lists of graphical rewrite rules saying what each does: "This one moves to the right. This one jumps over a rock. This one goes in the house. This one runs away..."
- Programming by demonstration solves the problem of how to specify the transformation from the left to the right side of a graphical rewrite rule. Some systems require the user to specify the transformation in a traditional programming language such as Lisp. Some systems try to infer the transformation using AI techniques, although no one has developed a general method for doing so. Some systems sacrifice generality in favor of simple transformations. Programming by demonstration has proven to be sufficient to specify powerful transformations and yet remain manageable by ordinary people.

Building an Ocean Simulation

Let's illustrate KidSim by building a simulation of marine life in the ocean. We'll create some fish, get them to move around, grow old, mate, have baby fish, get eaten by sharks, and, if they make it that far, die of old age. It takes about 30 minutes to create such a simulation in KidSim. We hope to convince our readers that it is well within their capability to do all the steps necessary.

First we create a new object in the copy box by clicking on the "tube of clay" symbol next to it. This squeezes out a "lump of clay", i.e. a new object, into the copy box. The idea is that out of clay, you can make anything. The drawing editor automatically appears so that we can make the new object look like something other than clay. We'll make it look like a clown fish:

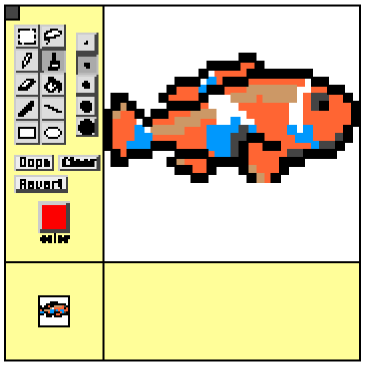


Figure 2 - the KidSim drawing editor

Once we have a drawing we like, we close the drawing editor window, and the object in the copy box appears as a clown fish. We now drag out a couple of fish and scatter them around the game board.

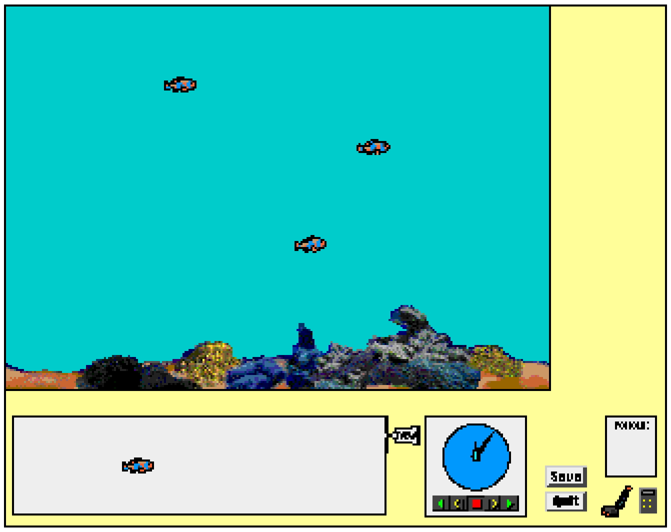


Figure 3 - the game board and copy box showing clown fish

We can run the simulation by clicking on the Run button under the clock: ▶. But when we do, nothing happens because we haven't defined any behavior for the fish. So now let's get to the heart of KidSim: how to program the behavior of simulation objects. We click on one of the fish in the game board. KidSim displays a "viewer" for that fish. This shows its rules (there aren't any), traits (symbolic data), and buttons.

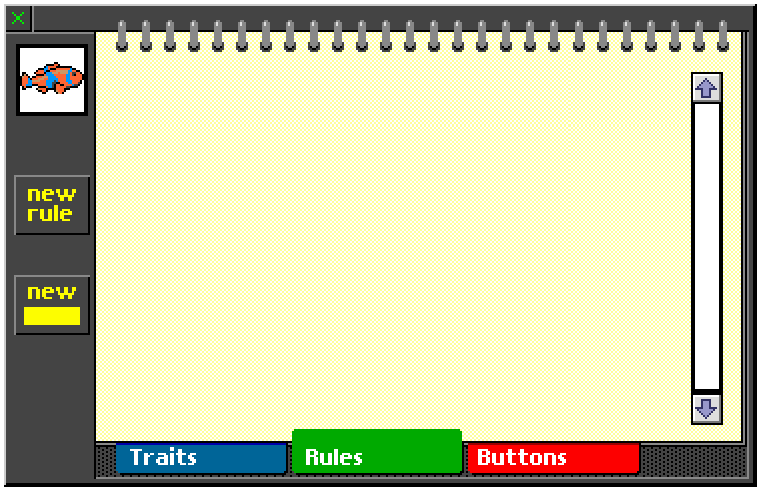


Figure 4 - the rule area in a viewer

We click on the New Rule button to begin defining a new rule. This causes several things to happen. (a) KidSim enters "record mode." (b) The rule editor appears, as shown in the lower right corner of figure 5.
(c) The game board darkens and shows the discrete spaces into which it is divided. This provides feedback that the system is in record mode and helps children communicate their intentions to the computer. (d) The area around the fish being modified is shown brighter. We call this area the "spotlight"; it focuses the user's attention on the object being dealt with. We can adjust the size of the spotlight by dragging the small handles provided. The spotlight defines the area of the game board with which this rule will deal. The area in the spotlight appears in both the left and right sides of the rule in the rule editor.

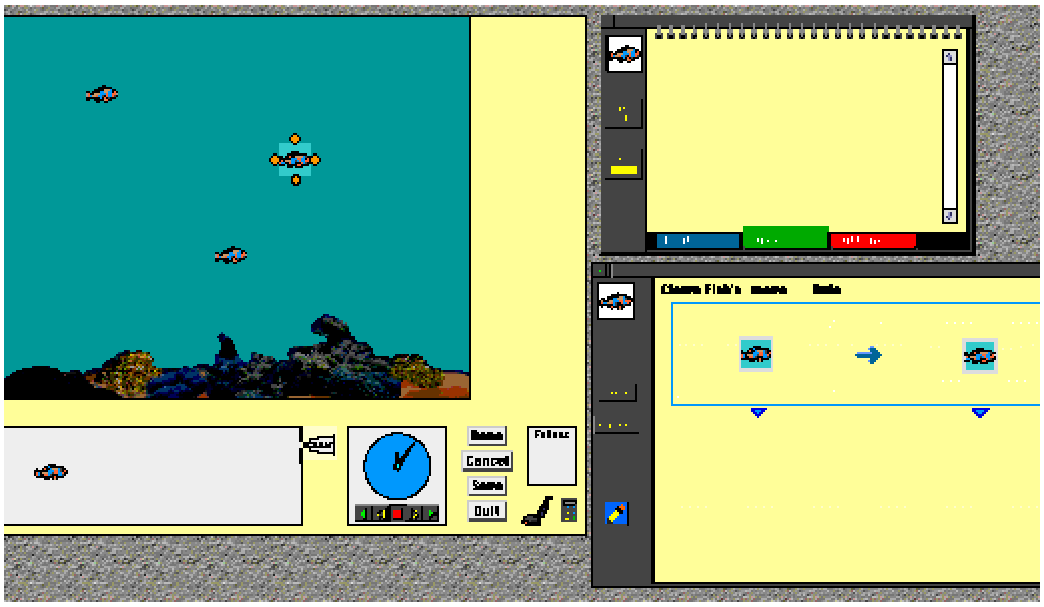
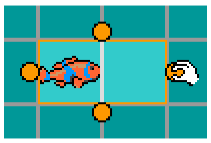
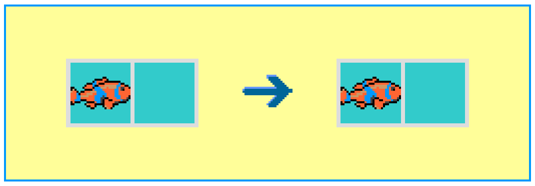


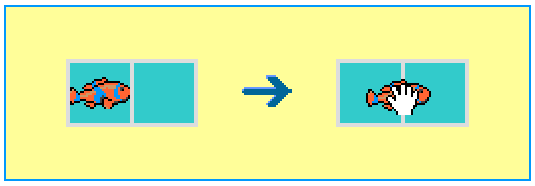
Figure 5 - KidSim in record mode

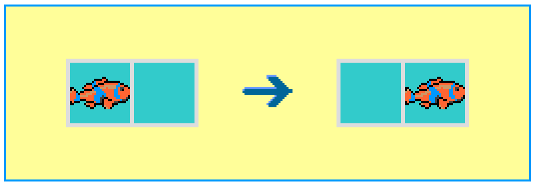
Now, any actions we do on objects in the spotlight or in the rule editor or on properties in a viewer will be recorded as part of the program for this rule. Let's make a simple "move" rule. First we'll make the spotlight include the space to the right. Any change to the spotlight size is reflected in the rule:





Then we'll edit the right side of the rule. Since all we want to do is move the fish to the right, we simply grab the fish with the mouse and drag it into the square to the right:





Then we click on the Done button. That's all there is to it. Nowhere did we type IF-THEN-ELSE, or BEGIN-END, or semicolons, or any other language syntax, and yet we've modified the fish's behavior. We edited the right side of the rule until it looked the way we wanted it. This is the essence of programming in KidSim:

programming by direct manipulation editing. The system records the editing actions in the program for this rule. This graphical rewrite rule may be read as follows: "if a clown fish has an empty space to its right, then move into that space."

Now when we start the clock, the clown fish we just modified and *all* clown fish on the game board start moving to the right. The reason all clown fish move is that we have found that children expect that things that look the same will behave the same. Thus we have changed our rule paradigm from associating rules with individual objects to associating rules with classes of objects. The "class" object is the one in the copy box. Copies made of it are instances of that class.

The KidSim convention is that on every clock tick, every simulation object on the game board gets a chance to run. KidSim starts at the top of each object's list of rules, tries each rule in order, and executes the first one that matches. As soon as one rule executes, that object's turn is over. On its next turn, KidSim starts over at the top of the object's rule list. Thus iteration is built into KidSim's rules. We have already seen that conditionals are also built into them: IF the left side of a rule matches, THEN execute the recorded program used to produce the right side.

Now let's create some other things that occur in the ocean: some other types of fish, some sharks, and some kelp. We do this just as before, by creating new objects (lumps of clay) and then drawing their appearance. Then we drag out a few copies of each:

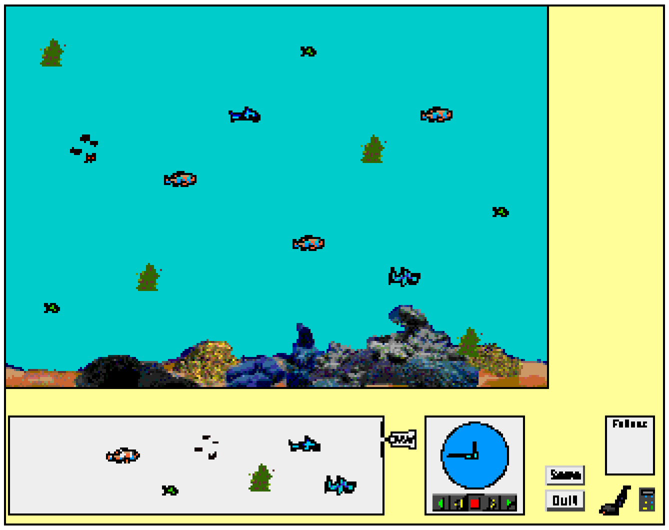


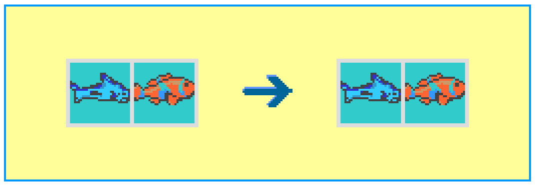
Figure 6 - an ocean simulation with several types of objects

When we run the simulation now, we see that only the clown fish move. We have not yet given the new objects any behavior. We define "move" rules for them as we did for the clown fish. If we want another fish to have exactly the same "move" rule as the clown fish, we can drag the "move" rule out of a clown fish's viewer and drop it on the other fish. KidSim will copy the rule and insert it in the destination object's list of rules. This makes it easier to spread behavior around. Here, for variety, we'll make each type of fish swim differently; some will swim up and down, some diagonally, and some the opposite direction. This will increase the likelihood that fish will encounter each other. (These steps are not shown.)

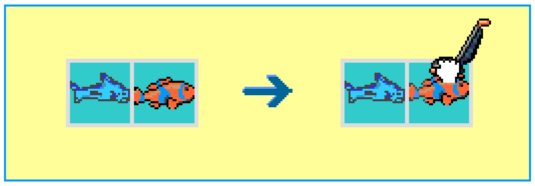
Sex and Violence in KidSim

Now we've got a bunch of fish swimming around in our "ocean," but nothing really exciting is happening. Obviously what we need is some sex and violence.

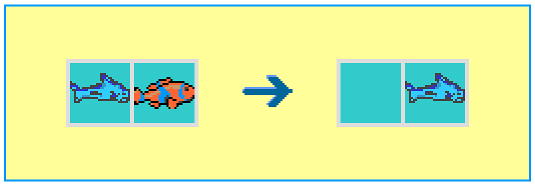
Being Americans, we'll start with the violence. Let's have sharks eat other fish. We'll define an "eat" rule for sharks. First we put a blue shark next to a clown fish, where it can reach it. We click on the shark to display its viewer and then click on its New Rule button to create a new rule and put the system into record mode. We expand the spotlight to include the clown fish:



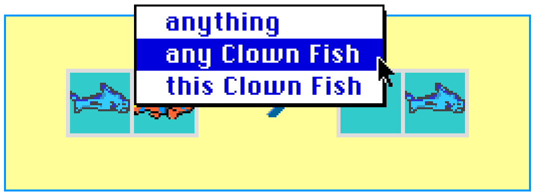
This enables the shark to detect a clown fish when it is right in front of its nose. If we wanted the shark to be able to detect it when it was farther away, we could expand the spotlight and place the clown fish farther away. To get it to eat the clown fish, we "vacuum it up" with the Vacuum Cleaner. The Vacuum Cleaner is the way we delete things in KidSim. We drag it over and drop it on the clown fish:



Now the clown fish is gone. (Actually it's still in the system, in the Vacuum Cleaner's bag. We can drag it back out later if we choose.) At this point, what we really want is to create a blood-curdling animation of a shark ripping into the clown fish's flesh with appropriate bone crushing sounds, and it will eventually be possible for a child to define such an animation; but it's not in KidSim yet, so we'll skip that part. The last thing we'll do is move the shark into the space occupied by the clown fish. The illusion is that the shark eats the fish:



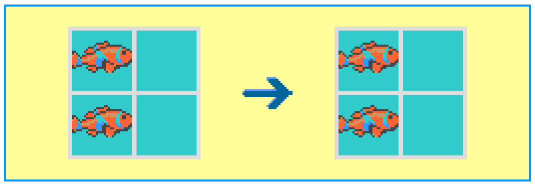
Actually, we want sharks to be able to eat any kind of fish, not just clown fish. If we click on the clown fish in the left side of the rule, a pop-up menu appears showing its type classification.



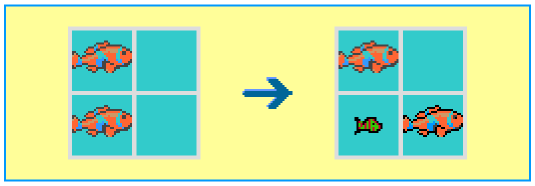
We used to allow an arbitrarily deep type hierarchies--e.g. clown fish, fish, ocean dweller, animal, living thing, object,--but we found that children couldn't construct these hierarchies. The process was too complex. They could *understand* class hierarchies, but they couldn't *create* them. Try as we might, we couldn't find a simple way for children to do it. So we eliminated them. Now every object has a fixed three-level hierarchy: this particular clown fish (instance), any clown fish (class), or anything (object). There is a way to define additional classifications in KidSim, but that is beyond the scope of this paper. So for now, we'll just change the classification to "anything." This rule will now match any object, i.e. the shark can now eat any object that appears next to it. (While this is not exactly what we want, it's actually not that unrealistic, given what has been found in the stomachs of sharks over the years.)

We're done with this rule, so we click on the Done button. Now when we run the simulation, blue sharks start eating anything they come across. If we want hammerhead sharks to do the same thing, we can drag the blue shark's "eat" rule onto a hammerhead shark. At any rate, objects now start disappearing from the simulation. If we run it long enough, most objects will eventually get eaten. We need a way to generate more objects in order to maintain a steady state. Clearly it's time for sex!

Let's define a rule that causes clown fish to mate. Whenever two clown fish are close to each other:



the one on top will "mate" with the one below, producing a baby fish. We make birth happen by dragging one of the fish, say the bottom one, out of the way, and then dragging a baby fish out of the copy box:



Just as vacuuming is recorded as a delete action, dragging from the copy box is recorded as a create action. Every time this rule executes, a new baby fish will be created. (Somewhat easier than what people go through.)

Now we have clown fish fighting back, creating new baby fish to counteract the sharks' eating. The only trouble is that a baby fish is not a clown fish. We need to make baby fish grow up and mature into clown fish.

The first thing we'll do is give baby fish an "age" property. We turn to the "traits" (properties) page of the baby fish's viewer. We create a new property in it by clicking on the New Property button. We'll call the new property "age" and give it a value of "1":

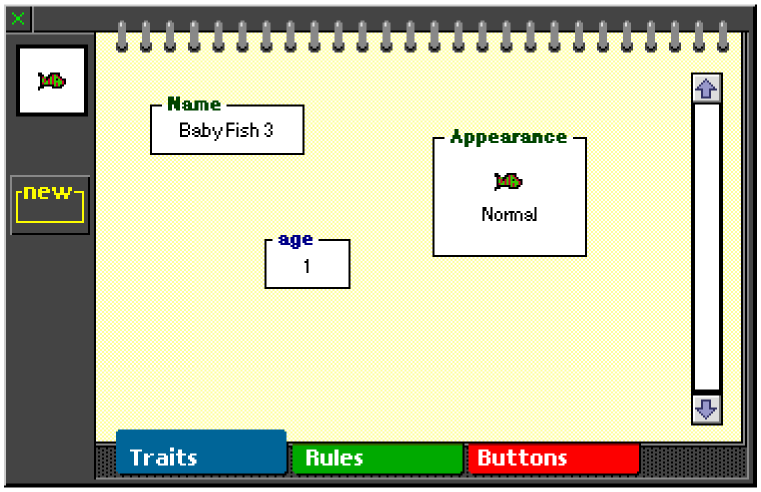


Figure 7 - the "traits" (properties) area in a viewer

To KidSim, numeric properties have no units; but we may think of numbers as being in any units we please, whatever seems appropriate for the current simulation. Let's think of a fish's age in terms of weeks. Properties in KidSim are equivalent to instance variables in traditional object-oriented programming languages. We've modeled their appearance after the boxes in Boxer [5], a language which has proven to make it easier for children to learn about variables.

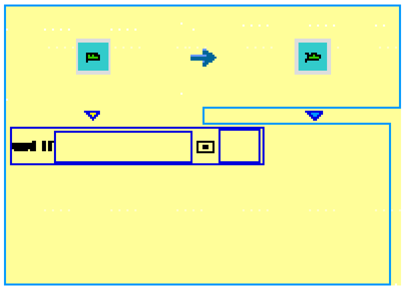
First we need a rule that increments the baby fish's age on every clock tick. As before, we create a new rule for it by clicking on the New Rule button. This time we aren't interested in the matching part of the rule; we want the rule to always match. So we leave the spotlight alone, just big enough to cover the baby fish. What we want is to record a change to the baby fish's age property, namely we want to increment it by one. Since the

system is in record mode, anything we do to an object's properties will be recorded. So all we have to do is change the "1" in the age property to "2". Ordinarily KidSim would record "put 2 into baby fish's age." But we have found that incrementing and decrementing by one are the most common arithmetic operations on numeric properties, so as a heuristic, KidSim instead records "add 1 to baby fish's age." We can change this if we don't like the choice, but here it's what we want. More complicated arithmetic calculations involving multiple properties from multiple objects in multiple steps can be performed with the Calculator.

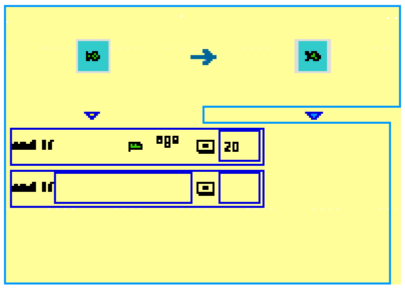
Since all we wanted to do was change the age property, we're done with this rule, and so we click on the Done button. Now on every clock tick, every baby fish's age property is incremented by one. That is, all baby fish now grow older.

The alert reader will notice a problem. Since an object's turn is over as soon as any rule matches, and since a baby fish's aging rule will always match, all baby fish will be able to do is grow older. They won't be able to move or eat or avoid sharks or anything else. However, there is a way in KidSim to mark rules of this sort so that they do not stop the rule interpreter. That is, a marked rule may match and execute, and the rule interpreter will still continue testing other rules. (The marking process will not be described here.) We'll mark the aging rule. Now baby fish will not only age but perform other actions as well.

The final thing we have to do is have a baby fish turn into a real clown fish when it is old enough. So we'll create a rule to do that. (That's the universal answer to solving problems in KidSim: *create a rule*.) We click on the baby fish's New Rule button, which puts the system in record mode. Again, we aren't interested in the pattern matching part of the rule, so we leave the spotlight alone. But this time we want to test one of the baby fish's properties. Specifically, we want this rule to apply only when the baby fish becomes 20 (weeks) old. We click on the triangle under the left side of the rule, causing a property test area to appear:



We drag the baby fish's age property into the left side of the test and type 20 into the right side. We then choose "=" from the pop-up menu in the middle:



This rule may be read as follows: "if the baby fish is on the board, AND IF the baby fish's age property equals 20, then...." Now we have to tell it what to do. What we want is for it to turn a baby fish into a clown fish. First we drag a clown fish out of the copy box. Then we copy any data we want to preserve, such as its name, into the clown fish. We do this dragging properties or property values from the baby fish's viewer into the clown fish's viewer. Finally we vacuum up the baby fish. The effect is that the baby fish becomes a clown fish. Lastly, we click on the Done button to end the rule.

Now, as we run the simulation, clown fish periodically encounter each other and spawn baby fish. If we start with enough clown fish, they should be able to keep up with the rate that sharks eat them, producing a steady state population. If we start with too few, they may all get eaten.

We can go on from here and investigate a variety of possibilities. For example, what happens if we make sharks able to detect fish farther away and thus eat more efficiently? What happens if we make clown fish have two babies, or ten, instead of one? What happens if we make the babies mature faster or slower? What happens if we give clown fish a defense mechanism, e.g. hiding in sea anemones as in real life? What happens if we add an energy level to sharks and fish so that they have to eat regularly or die? We might have to create plankton for the smaller fish to eat and set up a food chain. We can let our imaginations drive the simulation in directions that, while possible with conventional programming languages, were not practical before.

Summary

What we've accomplished in this exercise is to create

- an ocean simulation with several kinds of objects,
- fish which move around,
- sharks which eat them,
- clown fish that mate and have babies which grow up to be adult clown fish,

...in short, a simple ocean ecology. We can populate the simulation with any number of our objects, and we can observe the results. We can use the simulation for a variety of further explorations.

The point is that this is a moderately interesting and flexible simulation that was created by actions that you the reader are perfectly capable of doing. At least we hope you feel that way. It did not require any long learning time or studying a manual. And we hope you feel that children and their teachers could, for the first time, create such simulations.

References

1. Apple Computer, Inc. *Macintosh Human Interface Guidelines*, Addison-Wesley, Reading, MA, 1992.

2. Cypher, A. and Smith, D.C., KidSim: Programming Simulations by Demonstration, to appear in *CHI `95 Conference Proceedings*, ACM Press, New York, 1995.

3. Cypher, A., ed. Watch *What I Do: Programming by Demonstration*, MIT Press, Cambridge, MA, 1993.

4. Davis, R. and King, J. *An Overview of Production Systems*, Computer Science Department Report No. STAN-CS-75-524, Stanford University, 1975.

5. diSessa, A.A. and Abelson, H. Boxer: A Reconstructible Computational Medium, in Soloway, E. and Spohrer, J. *Studying the Novice Programmer*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1989, 467-481.

6. Newell, A. and Simon, H.A. *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

7. Rychener, M.D. *Production Systems as a Programming Language for Artificial Intelligence*, Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, 1976.

8. Smith, D.C., Cypher, A. and Spohrer, J., KidSim: Programming Agents Without a Programming Language, *Communications of the ACM*, 37, 7 (July 1994), 55-67.

9. Smith, D.C., Irby, C., Kimball, R., Verplank, W. and Harslem, E. Designing the Star User Interface. *Byte*, 7,4 (April 1982), 242-282.

back to ... Publications ← Allen Cypher