Cypher, Allen and Smith, David C. "KidSim: End User Programming of Simulations". To appear in *Proceedings of CHI*, 1995 (Denver, May 7 - 11). ACM, New York, 1995.
Copyright 1995 by the Association for Computing Machinery, Inc. Copying without fee is permitted provided that the copies are not made or distributed for direct commercial advantage and credit to the source is given. Abstracting with credit is permitted. For other copying, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923; Phone: (508) 750-8400, Fax: (508) 750-4744. For permission to republish write to: Director of Publications, Association for Computing Machinery. To copy otherwise, or republish, requires a fee and/or specific permission.

KIDSIM: END USER PROGRAMMING OF SIMULATIONS

Allen Cypher and David Canfield Smith

Advanced Technology Group
Apple Computer, Inc.
Cupertino, CA 95014
cypher@apple.com, dsmith@apple.com

ABSTRACT
KidSim is an environment that allows children to create their own simulations. They create their own characters, and they create rules that specify how the characters are to behave and interact. KidSim is programmed by demonstration, so that users do not need to learn a conventional programming language or scripting language. Informal user studies have shown that children are able to create simulations in KidSim with a minimum of instruction, and that KidSim stimulates their imagination.

KEYWORDS: end user programming, simulations, programming by demonstration, graphical rewrite rules, production systems, programming by example, user programming.

INTRODUCTION
KidSim is an environment that allows children to create their own simulations. In schools today, students express their ideas through drawings, written descriptions, and perhaps video and multimedia. Simulations open up an additional dimension, since they enable students to express ideas about the way objects behave and interact.

The educational philosophy of constructivism states that children learn best when they actively create. So we wanted to provide a tool that would allow children to construct simulations themselves. In order to create a system that would be engaging and appealing to children, we decided that it would be unacceptable to require them to use a conventional programming language or scripting language. Therefore, one of the major aims of our project was to find a simple, natural way for children to program simulations. We wanted to provide a set of capabilities that was sufficiently rich and expressive that children could build an interesting and diverse set of simulations, but we were willing to trade off computational power in order to maintain ease of use.

Based on a variety of informal user tests, we settled on a design that allows children to create simple rules of behavior by demonstration. To express more abstract behaviors, such as becoming tired over time or only eating when one is hungry, we devised interfaces that use a mixture of demonstration and direct manipulation. The resulting system, with its hybrid of interaction techniques, has satisfied our goals of both ease of use and expressiveness.

I. THE COMPONENTS OF KIDSIM
Using KidSim, children create worlds that contain a variety of characters, which have rules, appearances, and properties. A rule defines what a character is to do in a particular situation. Appearances allow the character to change its visual appearance, and properties are used to maintain information about the character.

When the simulation is running, KidSim continually advances its *clock*, and on each tick of the clock, each character on the board is given a chance to act according to its rules. Figure 1 shows a typical KidSim screen.

A. Rules
Each character has a list of rules. When a character is given its turn to act, KidSim searches down the list for the first rule that is applicable to the current state of the board. This rule is then executed.

Creating a Rule
Simple rules in KidSim are created by demonstration. That is, the user performs by hand each of the actions that the rule is to perform. KidSim then generalizes these actions to create a rule that will automatically perform those actions whenever it is executed.

The simplest rules to create in KidSim are rules which cause a character to move. For instance, Figure 2 shows how to create a rule which causes a character to move one square to the right.

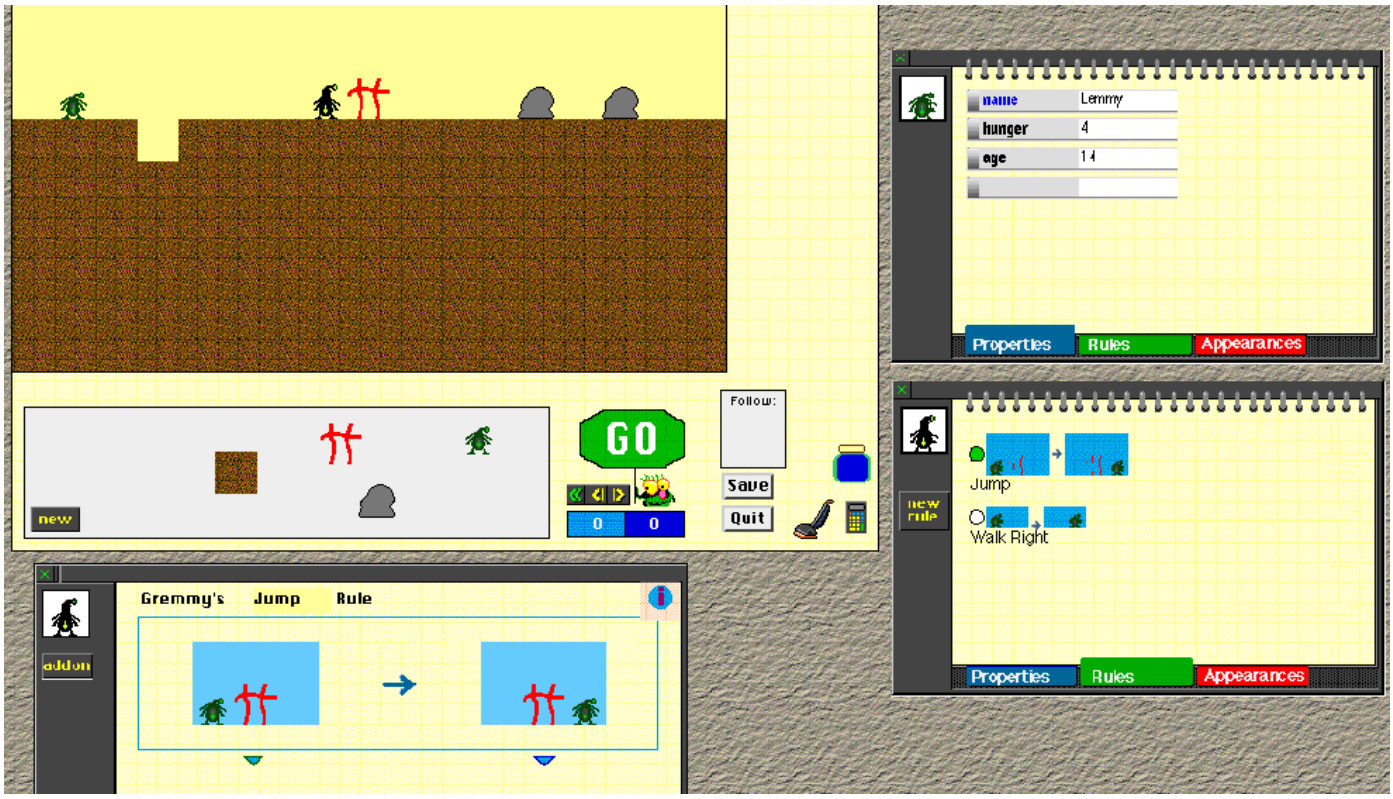


Figure 1. The KidSim screen. The window in the upper left corner contains the Board, with the Copy Box below it. On the right, the user has opened notebooks for the two gremlins on the Board. The upper notebook is open to the Properties page, and the lower is open to the Rules page. At the bottom of the screen is the Rule Editor, which is displaying the "Jump" rule.

When the user clicks on a character's "New Rule" button, the entire board darkens, except for a *spotlight* around the character. The user reshapes this spotlight to specify the context for the new rule. For the example in Figure 2a, the spotlight has been enlarged to include the square to the right of the character. Next, the user demonstrates what the rule should do by moving the objects in the spotlight. In Figure 2b, the user moves the character into the square on the right, and then clicks on the done button.

The visual representation for a rule shows a picture of the "before" state of the rule on the left, a picture of the "after" state of the rule on the right, and an arrow between the pictures. The rule in Figure 2 can be read as "If there is an empty space to the right of me, move into it".

Since rules are tested on every tick of the clock, this one rule is sufficient to cause the character to move across the board when the simulation is running.

B. Appearances
A character can have a variety of appearances. By creating rules which change a character's appearance, the user can animate the character's motion, change the direction it is facing, change its facial expression, and so on.

Figure 3 shows a rule for jumping over fences. Note that the *actions box* on the right side of the rule shows a description of each action that will take place when the rule is executed. To create this rule, the user moves the character to the square above the fence, drags the "jumping" appearance into the *current appearance box* on the appearance page of the character's notebook, moves the character to the square to the right of the fence, and then moves the "normal" appearance into the character's current appearance box.

C. Properties
The features of KidSim described so far make it easy for children to create rules that cause their characters to move in different ways, depending on the objects around them. Since this is the essence of many simulations, we designed an interface which was specially tailored to this capability. However, simulations of this sort are limited in that the only way a character can be different from one time to the next is by its location on the board, and by its appearance. But interesting simulations often depend on the fact that characters change as the simulation progresses. They may get old, or sick, or stronger, or smarter. In order to allow for internal state that is not bound to the visual appearance of the character, KidSim characters have *properties*.

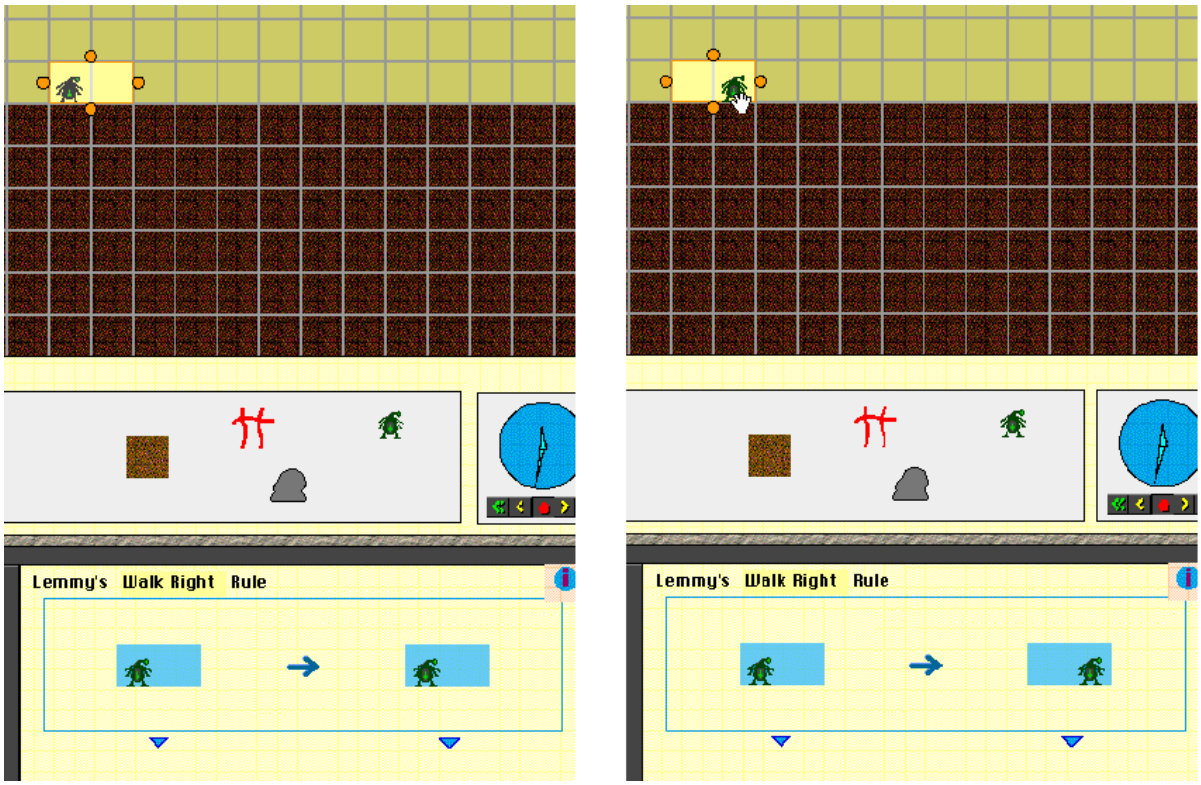


Figure 2. Creating a "Move Right" rule. The user shapes the spotlight to include the square to the right of the character, and then drags the character into that square. The rule, displayed at the bottom, shows the initial and final states for the rule.

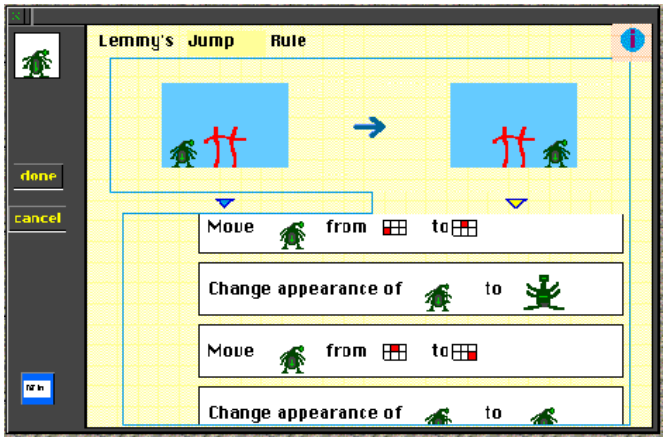


Figure 3. The "Jump" rule. The actions box has been opened to show each of the actions performed by the rule.

There are a few system-defined properties, such as *name* and *location*, but users can also create their own properties. For instance, a user could create the properties "age", "hunger", and "skills" for a character. Properties serve the role of instance variables in object-oriented programming.

Modifying properties
Figure 4 shows a modified "walk right" rule, where walking makes the character become hungry. The user creates the "hunger" property by typing "hunger" and "0" into the empty property box at the bottom of the list of properties. To add on to the "walk right" rule, the user clicks on the "Add On" button for that rule. This executes the actions that are already part of the rule, and then lets the user demonstrate further actions. In this case, the user clicks on the "0" in the "hunger" property, and changes it to "1". KidSim generalizes this demonstration to mean "Add 1 to my hunger". If this is not the intended interpretation of the demonstration, the user can edit the description to, for instance, "Put 1 into my hunger". KidSim includes a calculator so that the user can create more complicated rules.

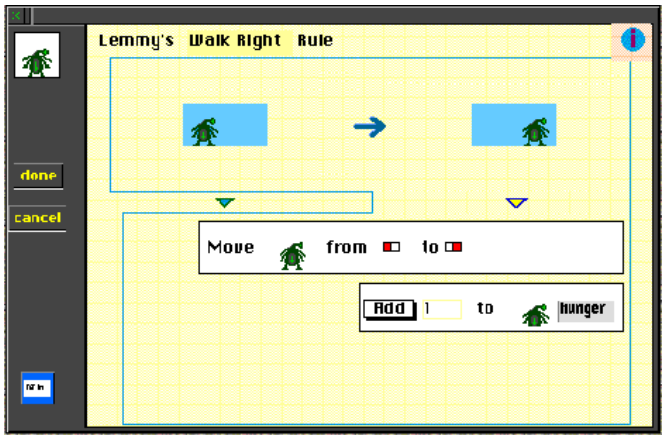


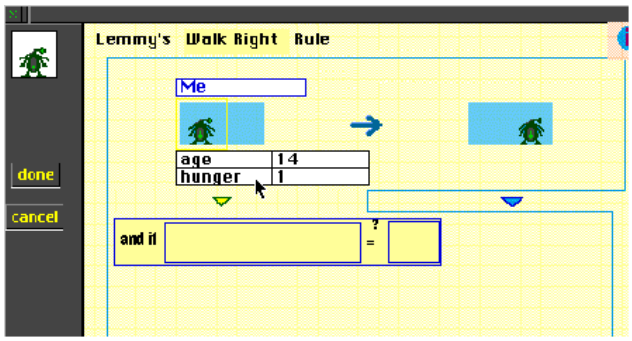
Figure 4. The "Walk Right" rule, modified to increment the character's hunger property.

Testing properties
Besides modifying property values, it is important that rules be able to test properties. The picture on the left side of a rule expresses conditions about the state of the squares neighboring a character, such as "If the square to the right of me is empty". The box below this picture (called the *conditions box*) is used to add further conditions to the rule.

Figure 5 shows a further modification of the "walk right" rule, which will prevent the character from walking if it is too hungry. To create this conditional, the user clicks on the character in the left part of the rule, drags the "hunger" property from the menu that appears, drops it into an empty conditional, types in "10" on the right side of the conditional, and chooses "<" as the test.

The final "walk right" rule can be read as "If there is an empty square to the right of me, and if my hunger is less than 10, then move into the square and add 1 to my hunger".

It should be noted that a rule can refer to the properties of *any* character.



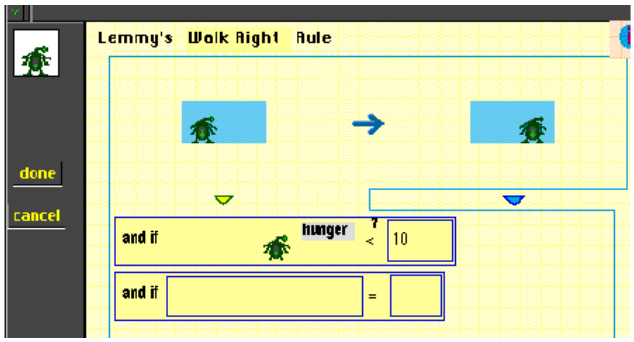


Figure 5. Creating a conditional. The user clicks on the character to bring up the list of properties and drags the hunger property into the empty conditional box. The user then types in the value "10", and chooses the "<" test from a popup menu.

D. Inheritance
The *Copy Box*, located below the board, displays the different types of characters in the current simulation world. The Copy Box contains character *types*, and the board contains character *instances*. When the user drags a character type from the Copy Box onto the board, an instance of that type of character is created on the board. The user can create new types of characters by clicking on the *New* button in the Copy Box.

Each type of character has its own properties, appearances, and rules. Every instance of that type of character will have the same properties, appearances, and rules. When the user adds a property, appearance, or rule to any instance of a character, all characters of that type also acquire that property, appearance, or rule. However, each character instance has its own *value* for a property. So, for example, if gremlins have an "age" property, each individual gremlin on the board may have a different age. Furthermore, each character can have its own drawing for each appearance. Thus, individual gremlins need not all look the same.

E. Rule Execution
On each tick of the simulation clock, every character on the board must be given a chance to act. Some simulation systems allow characters to act in parallel. That is, the characters all examine the current state of the board, and decide what to do based on that state. This scheme is consistent with the common sense notion of how independent entities behave in the real world. However, it can lead to confusing situations in a simulation. For instance, suppose two characters are standing on either side of an empty chair, and they each have rules stating "If I am next to an empty chair, sit in it." With parallel execution, both characters will notice that the chair is empty, and so when the simulation runs, they will both sit in the chair.

In order to avoid such situations, we adopted the less sophisticated model of sequential execution. At each tick of the simulation clock, the first character on the board (i.e. the one that was first placed on the board) examines the current state of the board and performs the first rule that matches this state. Then the second character examines the new state of the board and performs its first matching rule. This process continues until the last character on the board has been given a chance to act.

This scheme has proven to be quite understandable by our users. One problem with it is that the order in which pieces are placed on the board can affect the behavior of the simulation. Rather than trying to counter this side-effect, we simply provide user feedback to make the situation comprehensible: when the user steps the clock one tick at a time, KidSim highlights each character in turn as it is given a chance to act.

II. USER STUDIES

A. Early Studies

When we started to design KidSim, we conducted several informal user tests with fifth-grade children. In order to test whether graphical rewrite rules -- rules which show a "before" picture on the left, and an "after" picture on the right -- were a viable approach, we posed problems to the students, such as creating a character which could walk to the right and jump over obstacles. We asked them to draw before-after pictures on "Post-It" notes, and we then had them act out the roles of characters in the simulation, following the rules they had written. We were encouraged that the students were able to write rules in this format, and that they quickly understood how to test the rules against the current state of the "world". Furthermore, children were able to understand rules that had been written by other children.

In another informal study, four students created a video prototype of the KidSim board, in the style of Vertelney [14], using magnets under a piece of Plexiglas to move their characters. This gave us confidence that our use of a clock, with characters taking turns to act, was appropriate.

Even more informally, three fifth-grade teachers were early (and patient) users of the first working versions of our program. Even when the program was crashing quite regularly, they were able to construct a scenario where it started to rain, and characters went for shelter in a nearby house. Once the prototypes became somewhat more stable, we tested KidSim regularly on the students in their classes.

These early tests with teachers and students were invaluable for finding problems in specific user interface objects and interaction techniques. For instance, we originally used double-clicks on a character to bring up the notebook for the character, and single-clicks on buttons to create new rules. This inconsistency was confusing for all of our users, so we eliminated all double-clicks from our design. Some other examples: We found that a trash can was inappropriate for disposing characters, that allowing users to erase individual items in a conditional expression was excessive and unnecessary, and that adding the text "and if" at the beginning of each conditional expression made rules easier to understand.

These early tests also pointed out some more fundamental problems. Users would sometimes expect that there was only a fixed set of names that they could use for properties, as if the properties were predetermined by the system. We surmised that part of the problem was that the system properties, like "name", were listed together with the user properties. Although system properties are distinguished by displaying their names in blue, this distinction was evidently too subtle. Also, it seems that the visual appearance of properties is too heavy-weight, and that it implies that the computer is doing something special with them. We are therefore switching to a design that uses Boxer-style boxes [4], and that displays system properties in a separate area.

An encouraging result of our studies is that girls seem to enjoy using KidSim just as much as boys. We want to design a system that does not have a gender bias, and we are interested in conducting further studies to better understand which features of an interactive environment are particularly appealing to girls, and which are particularly unappealing.

B. The UK Study

Once we had created a working version of KidSim, with all of the basic features functioning as we intended, an informal user study of KidSim was conducted at the Centre for Research in Development, Instruction and Training at the University of Nottingham by Prof. David Gilmore. The study involved 56 children between the ages of 8 and 14. Their exposure to KidSim varied between 1 hour and 8 hours, in multiple sessions, with the children working predominantly in groups of 2 or 3. Most of the children claimed some computer experience, though generally not with Apple Macintosh computers. Minimal instruction in KidSim was provided, consisting of approximately 10 minutes with an introductory worksheet. Generally two researchers, who had about 2 weeks experience with KidSim, were present in the room at each session. A session consisted of 4 groups of students.

The sessions were quite open-ended. Initially, the students were given some ideas of rules to write, such as "move a creature rightwards along the ground". All sessions were video-taped and audio-taped.

Most all children found the rule-writing interface easy to use and were able to generate multiple rules for their characters. Furthermore, the system provoked their imaginations, and the children invented goals for themselves and created their own characters and situations. They created a soccer game, PacMan, a maze traversal game, ninja turtles, and an aquarium.

This study showed that children can create rules, and that they can read an individual rule. However, it was not clear from this study whether they can understand sets of rules, and how multiple rules and characters interact.

C. Design Changes

Our main design goal was to produce a tool that children would be able to use to create their own simulations. Every step in the design involved tradeoffs between making KidSim powerful enough that children would find it expressive and engaging, and making it simple enough that they would not find it frustrating or confusing. Our periodic user testing has helped us to see where our initial design choices erred in one direction or the other.

Simple Inheritance

In the original version of KidSim (see [12]), it was possible to create arbitrarily deep hierarchies of character types. For instance, one could create clown fish, which are a type of fish, which are a type of animal, which is a type of object. We wanted users to create new rules and properties by adding them to a particular character on the game board, but we were not satisfied with any of our schemes for determining how to propagate new rules and properties up the inheritance hierarchy. Furthermore, the deep hierarchy led to potentially confusing situations, since users could instantiate abstract superclasses. Thus, it was possible to have an object on the game board which was an instance of animal, while other objects were instances of clown fish and sharks.

As a result of these difficulties, we changed to a simple inheritance scheme that admits only a single level of character types. This mechanism is certainly less powerful. For example, to add a "swimming" rule to all clown fish, sharks, and whales, users must put a copy of that rule in each of these three character types.

Independent of the "deep hierarchy" issue, we have been interested in allowing children to create new categories at any time. After creating a rule to jump over fences, for instance, a user might want to use the same rule to jump over rocks. We therefore added a new feature to KidSim, called *Jars* (see Figure 6). The user can create a new jar, name it "Obstacles", and put Fences and Rocks in the jar. When the user clicks on a fence in a rule (see Figure 7), a popup menu lists "Fence 27", "Fence", "Object", and "Obstacles". The user can select any one as the desired interpretation of what that object is to represent in the rule. Thus Jars can compensate somewhat for the absence of a deep hierarchy.

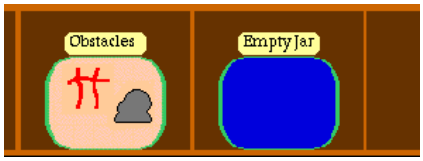


Figure 6. An "Obstacle" Jar, containing Fences and Rocks.

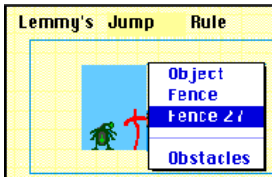


Figure 7. Selecting the desired generality for an object in a rule.

Characters larger than a square
Our current implementation assumes that every character fits into a single square on the board. This simplifying assumption makes it much easier to specify rules in terms of the squares neighboring a character. Although we were quite content with this simple approach, our users were not. They frequently want to create worlds where some characters are much larger than others. This means that our initial design decision resulted in a tool that was not sufficiently expressive. For instance, one user wanted to create a large horse that could carry several riders, and found it very unsatisfying to have to draw the horse in a single square. Therefore, our user studies have convinced us to modify KidSim to allow characters larger than a square, even though this will complicate the rule system.

III. KIDSIM AND END USER PROGRAMMING
Rule creation in KidSim uses of mix of Programming by Demonstration (PBD) and Direct Manipulation techniques. PBD is used during rule creation to show how a character moves, to change appearances, to create and delete characters (by dragging a character from the Copy Box into the spotlight, or by vacuuming a character), and to change the values of a property. Direct manipulation is used during rule creation to specify the context (by reshaping the spotlight), to generalize a character (by choosing the instance, the type, "Object", or a jar from a popup menu), to specify conditional tests (e.g. "my hunger < 10"), and to change the values of a property.

One of the reasons why programming is difficult is that programs are abstract. The strength of Programming by Demonstration comes from the fact that it allows the user to create an abstract rule by demonstrating what that rule should do in a specific situation.

KidSim rules are designed to bridge the gap between the specific and the abstract by conflating the specific example with the abstract rule. That is, the graphical representation of a rule has been designed so that the user can view this single representation in two different ways: 1) during rule creation, the rule should serve as a representation of the actions that the user has just performed, and 2) during rule application, the rule should serve as a representation of the general situation in which the rule is to apply, and of the general actions that are to occur when the rule is applied. For instance, while the rule in Figure 3 is being created, it should seem to the user that the fence in the rule refers to the fence to the right of the gremlin on the board. Later, when this rule is applied to a different gremlin confronting a different fence, it should seem natural to the user that the fence in the rule now refers to this completely different piece of fence.

As the user creates a rule, KidSim immediately abstracts the rule in a variety of ways, and the rule representation has been designed with the intent that none of these abstractions will confuse the user.
* **Instance abstraction.** For every character *instance* in the spotlight, the rule substitutes the character *type*. At creation time, since the appearance of the character type is likely to be quite similar to the appearance of the character instance on the board, this substitution should not create too large of a gap for the user to understand. And at application time, the appearance of the character instance on the board should be sufficiently close to the current appearance of the matching character in the rule that the user is able to understand that the characters do indeed match.
* **Motion abstraction.** When the user moves a character in the spotlight, the rule interprets this as a move *relative* to the starting location of the matching character. Users in fact expect this generalization, and have never expected the alternative interpretation that the move is to apply to the character instance with which the rule was originally created, or that the move is to this absolute location on the board.
* **Property abstraction.** When the user drags a property into the conditional box (on the left side of the rule) or the action box (on the right side of the rule), the rule generalizes this to refer to the property value, at the time of execution, of the matching character. At application time, users should not expect the original value from the time of rule creation to be used.
* **Property change abstraction.** When the user changes the value of a character's property, the rule displays a generalization that will produce the same change in the future (such as incrementing the value by 1). At creation time, users should not be surprised by the corresponding expression that appears in the rule.

Our user studies have shown that our rule representation is generally successful in serving its dual roles. However, the studies showed the last "should" mentioned above (for property changes) to be incorrect. In the original version of KidSim, rules to change the values of a property could only be created by demonstration. And some users were indeed confused during rule creation when, upon changing the value of a property, a statement of the form "Add 1 to my hunger" appeared in the rule.

In response to our user tests, we added a feature so that users could create the property-change expression by hand: the user selects "Put, Add, or Subtract" from a popup menu, types a number in the space to the right, and then drops in a property, to create a statement such as "Add 1 to my hunger". When we showed this new interface to one user, she found it understandable and was able to use it. We then explained how the same statement could be created by demonstration. She was now able to understand immediately this procedure which had previously confused her, and went so far as to state that it was much easier than creating the expression by hand. She now uses the demonstrational approach exclusively. We are planning to add a "recording agent" animation to KidSim, which will appear on the screen, watch the user change a property value, and then explicitly add the appropriate property-change expression to the rule. Perhaps with this additional feedback, the demonstrational approach will be easier for first-time users to understand.

Computer Literacy
Alan Kay has argued that there are powerful, yet difficult, concepts in programming, and that it is important that end user programming does not give up on teaching these concepts, in favor of making things easy [8]. Our intent is to remove the unimportant difficulties (like syntax) that unnecessarily impede students, so that they can proceed to work with the important concepts (like abstraction).

The main value of KidSim lies in an easy-to-use interface that allows children to create a fairly rich set of rules of behavior. Although KidSim is not a complete programming language -- for instance, its control structure is limited to a fixed sequence for testing rules -- it nonetheless embodies some important concepts in computer programming.

One of the advantages of KidSim's approach to programming is that it has eliminated many of the syntactic and semantic problems that confront users of conventional programming languages and scripting languages. Users need not worry about whether statements must end with a semicolon, or whether the word "the" is needed in an expression.

By working with KidSim, users can learn about the step-by-step execution of commands, about simple inheritance of properties and rules, about variables (i.e. properties), and about abstraction in rules.

IV. RELATED WORK

KidSim draws on four traditions in computer programming and human-computer interaction: production systems, graphical rewrite rules, programming by demonstration, and simulations. Executing the first matching rule in a list of test-action rules comes from production systems [3]. Graphical rewrite rules are used as a programming language in BITPICT [6], and are used to program simulations in Tableau [7], ChemTrains [1], and AgentSheets [11]. Mondrian [10] uses programming by demonstration to create rules represented by before-after pictures, similar to the storyboard representation of Chimera's macros [9]. The mixed icons and text used to represent program steps was used in Shoptalk [2].

KidSim is a successor to Playground [5]. Playground had the same goal of enabling children to create simulations. The most important positive thing that we learned from Playground was that its basic model of allowing children to create their own characters, and to attach rules to characters, was powerful and engaging. The most important negative thing that we learned was that scripting languages are too hard for children, even though Playground characters had a nice structure for storing programs, and the system provided a structure editor for creating syntactically correct statements. We also learned that characters should be allowed to directly manipulate other characters, since Playground did not allow this, and users found it too restrictive.

The system closest to KidSim is AgentSheets [11]. Agent-Sheets is a general-purpose simulation environment, which experienced developers can use to create a variety of domain-specific applications. That is, AgentSheets itself is not intended for end users, but it provides a set of tools that programmers can use to produce a great variety of systems tailored to specific end users. Notably, Repenning produced a domain-specific application for Turing machines that uses graphical rewrite rules, with end users creating Turing machine programs by demonstrating the steps in the program [11, pp. 80 - 82].

LiveWorld [13] is similar to KidSim in the style of the simulations that its users can create. Users make objects that move around on a board and interact with nearby objects. However, all programming in LiveWorld is done in Lisp, and therefore it is a much more powerful environment. Also, LiveWorld characters can contain sensors, which are a general and very effective means for specifying regions and objects of interest in a rule. LiveWorld also employs a powerful inheritance scheme.

Pinball Construction Set and SimCity are games that allow users to create interesting simulations, but users are limited to the pre-programmed behavior built into the objects that come with the application. KidSim is more akin to Rocky's Boots, which lets users create simple programs from a small set of primitives.

V. CONCLUSION

Future Directions

There are several features that we are planning to add to KidSim. We would like rules to be able to produce sounds, and to be able to test for sounds. We plan to introduce a feature to rules for referring to objects that are not spatially close to the character. We want to add buttons and switches, so that users can interact with a simulation as it is running, as in video games. And we are adding subroutines, so that rules can be grouped together. Subroutines should help to manage the complexity of having large numbers of rules, and also provide some additional control over the order in which rules are tested.

We would like to conduct further user tests to see whether children are able to use properties effectively, and whether they can understand large sets of rules. We would also like to determine whether KidSim is suitable for younger children.

Summary

The KidSim environment enables children to create their own simulations, and it encourages them to explore dynamic interactions between objects. Through programming by demonstration, children are able to create rules without using a conventional programming language. We have described how a process of prototyping and iterative design with informal user studies has helped us to build a system which is easy for end users to use, and which provides them with an expressive medium.

Acknowledgments

Warm and special thanks go to the teachers who motivated us, struggled with our early prototypes, and helped design KidSim: Betty Jo Allen-Conn, Julaine Salem, and Phyllis Lewcock.

We would like to thank Alan Kay for many important ideas. His respect for teachers, for the art of teaching, and for the importance of powerful ideas has informed our work.

Thanks to Jim Spohrer, David Maulsby, Edwin Bos, Kurt Schmucker, Stephanie Houde, Jeff Bradshaw and Rachel Bellamy for many valuable discussions.

We appreciate the following people for advocating important features: Alex Repenning for sequential execution, Enio Ohmaye for jars, and Peter Jensen for simple inheritance.

Many thanks to the programmers who have assisted us: Peter Jensen for the port to Prograph, Rodrigo Madanes for appearances, Dave Vronay for the drawing editor, David Maulsby for jars, Edwin Bos for subroutines, and Don Tilman and Ramón Felciano.

For his strikingly creative animations and graphics, we thank Mark Loughridge.

Thanks to the SK8 team. KidSim wouldn't exist without this wonderful environment: Ruben Kleiman (architecture), Adam Chipkin (scripting), Hernan Epelman-Wang (graphics), Brian Roddy (interface), and Alan Peterson (support).

For management and support, we thank Dana Schockmel, Kurt Schmucker, Jim Spohrer, Mark Miller, and Rick LeFaivre.

For the UK user test, we thank David Gilmore, Karen Pheasey, and Jean and Geoff Underwood. Their study was funded by the UK Economic and Social Research Council and NATO. And finally, thanks to all of the students who tested KidSim.

REFERENCES

1. Bell, B. & Lewis, C. ChemTrains: A Language for Creating Behaving Pictures. In *1993 IEEE Workshop on Visual Languages*. Bergen, Norway, 1993, pp. 188-195.

2. Cohen, P. Synergistic Use of Direct Manipulation and Natural Language. In *Proceedings of CHI '89*. ACM, New York, 1989, pp. 227 - 234.

3. Davis, R. & King, J. An overview of production systems. Rep. STAN-CS-75-524, Computer Science Dept., Stanford Univ., Stanford, CA, 1975.

4. diSessa, A. A principled design for an integrated computational environment. *Human-Computer Interaction,* 1, 1985, pp. 1 - 47.

5. Fenton, J. & Beck, K. Playground: An object-oriented simulation system with agent rules for children of all ages. In *Proceedings of OOPSLA '89*. ACM, New York, 1989, pp. 123 -137.

6. Furnas, G. New graphical reasoning models for understanding graphical interfaces. In *Proceedings of CHI '91*. ACM, New York, 1991, pp. 71 - 78.

7. Kay, A. Tableau, unpublished manuscript.

8. Kay, A. Foreword to Cypher, A. (Ed.), *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1993.

9. Kurlander, D. & Feiner, S. A History-Based Macro by Example System, In Cypher, A. (Ed.), *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1993, pp. 323 - 340.

10. Lieberman, H. Mondrian: A Teachable Graphical Editor, In Cypher, A. (Ed.), *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1993, pp. 341 - 360.

11. Repenning, A. AgentSheets: A tool for building domain-oriented dynamic, visual environments. Ph.D. dissertation, Dept. of Computer Science, University of Colorado at Boulder, 1993.

12. Smith, D.C., Cypher, A. & Spohrer, J. KidSim: Programming Agents Without a Programming Language. *Communications of the ACM*, 37(7), July 1994, pp. 54 - 67.

13. Travers, M. Recursive Interfaces for Reactive Objects, In *Proceedings of CHI '94*. ACM, Boston, 1994, pp. 379 - 385.

14. Vertelney, L. Using Video to Prototype User Interfaces. *SIGCHI Bulletin*, 21(2), October 1989, pp. 57 - 61.

---

back to ... Publications ← Allen Cypher