Programming Substrates to Create Interactive Learning Environments

Alex Repenning

University of Colorado

Abstract

The design of an effective interactive learning environment requires understanding the intricate relationships among people, tools, and problems. Many end-users do not have the necessary skills, nor the time or patience to compose programs from computer science-sanctioned programming primitives. End-users require environments that elevate the task of programming to the manipulation of components that are directly pertinent to the problems to be solved. This paper introduces the Agentsheets programming substrate employed by *designers* to create *interactive learning environments* that are geared toward *end-users* solving specific problems. A number of educational and industrial applications are used to illustrate the design and use of Agentsheets environments in domains such as art, artificial life, environmental design, games, kitchen design, and visual programming.

1. INTRODUCTION: PROGRAMMING SUBSTRATES CREATE LEARNING OPPORTUNITIES

This article describes the Agentsheets programming substrate that is used to create Interactive Learning Environments (ILEs). The long-term vision is to create an Agentsheets project called "SimCity‰ in 10 minutes." This ambitious goal may never be achieved but it motivates research, design, and implementation of new programming approaches that empower users to create SimCity-like applications in a very short period of time. We do not focus on a single programming approach, but instead explore different approaches and analyze what kind of problems they are suited for.

Agentsheets helps *designers* to build ILEs, such as end-user programming environments, domain-oriented visual programming environments, simulation environments, and design environments, for *end-users*. Designers and end-users of ILEs have the following *learning opportunities*:

- **Learning through Programming**. In the spirit of constructionism a programming substrate should enable people to construct personally meaningful products (Resnik, 1992). By doing so people can learn about domains such as mathematics through programming (Papert, 1980). The challenge for a programming substrate is to minimize accidental complexity of the programming task. That is, if the objective is to learn about a problem domain then the complexity faced when constructing a program should reflect the intrinsic complexity of the problem domain and not the complexity of formalizing the problem domain through programming. For instance, the complexity arising from building a SimCity-like application should be driven by the complexity of the city simulation model employed and not the intricacies of a programming substrate used to implement the model.
- ***Learning about Programming***. Agentsheets supports learning about programming principles such as object-oriented programming, parallel programming, and user interfaces. This type of learning is sometimes considered a side effect of doing the "real" thing, but for computer science majors and, more generally, for people interested in programming this learning experience may be important.
- ***Learning by Using***. A programming substrate should allow people to create engaging applications that are fun and interesting to use. For instance, programming substrates should facilitate real-time animation, interaction, and sound. The application created should not be just a

I thank the HCC group at the University of Colorado, Gerhard Fischer, and Clayton Lewis, who contributed to the conceptual framework and the systems discussed in this paper. This research was supported by the National Science Foundation under grant No. RED-9253425, Apple Computer Inc., and U S West Advanced Technologies Division.

Correspondence and requests for reprints should be sent to Alex Repining, University of Colorado, Department of Computer Science, Campus Box 430, Boulder, CO 80309.

"toy" application that will be thrown away once the lesson is learned. A programming substrate that allows students to implement a quick-sort algorithm may be an effective way to learn through programming but unless the student has to sort interesting numbers very little is to be learned by actually using the application created.

These learning opportunities are instances of *learning through problem solving*. The problems to be solved can be related to creating or using an ILE: How should a car be programmed to behave in certain ways in a SimCity like environment, or where should traffic signals be introduced to minimize traffic accidents at a dangerous intersection. In both cases it is important to understand how tools are used by people to interact with some problem domain. This paper analyzes the relationships among people, tools, and problems, introduces the Agentsheets programming substrate; illustrates how Agentsheets is used to create ILEs; and finally reports on the experience of users creating ILEs.

2. PEOPLE, TOOLS, AND PROBLEMS

To build programming tools that are effective interactive learning environments, we need to understand the intricate relationships among people, tools, and problems. Guttag (1991) summarized the difficulty of programming by pointing out that "the wrong people are using the wrong methods and the wrong technology to build the wrong things." **What** people are using **what** kind of tools to solve **what** kind of problems? This paper discusses the use of *substrates to build programming tools* to improve the relationships among people, tools, and problems.

The needs of end-users typically cannot be satisfied with general-purpose programming languages because end-users either do not have the necessary skills, the time, or the patience to compose programs from classical computer science programming primitives such as iterations, sequences, and branching (Lewis & Olson, 1987). *Usability, domain-orientation,* and *control* are three important characteristics describing the relationships among people, tools, and problems (Figure 1):

- ***Usability*** *(People and Tools):* How easy is it to understand and use a tool? Usability does not imply usefulness. A tool can, at the same time, be usable but not useful. A hammer, for instance, is usable by many people, yet it is not very useful in bread baking. Usability does not make an assessment about how *useful* a tool is with respect to a problem to be solved.
- ***Domain-Orientation*** *(Tools and Problems):* Tools are oriented toward problems, but problems are also oriented toward tools. The problem of assembling a car quite naturally leads to the use of traditional tools such as screwdrivers. The fact that screwdrivers are useful tools to assemble a car is not accidental. These tools have been known for a very long time and were taken into account when the parts of the car were designed.
- ***Control*** *(People and Problems)*: If the problem is to get bread, then baking the bread from scratch, using a bread machine, or buying the bread at a bakery are three different approaches leading to a similar result. However, these approaches differ considerably with respect to how much *control* a person has over the process of creating a solution and how much effort is involved.

The following sections elaborate the three relationships that will lead to design principles for programming substrates.

2.1. Usability and Domain-Orientation

If a tool is easy to use (high usability) and if the tool's functionality is relevant to the problem domain (high domain-orientation) then a tool is *useful* to solve a problem. In this paper I limit the discussion on usability to the *use of visualization*. By use of visualization I mean the use of effective perceptual information (Gaver, 1991) to design a visual representation.

Programming approaches can be characterized along the dimensions of *domain-orientation* and *use of visualization* (Figure 2). Visualization is a syntactic characteristic of a programming language in the sense that it renders the look of concepts but not their meaning. The use of visualization is presumed to simplify programming by capitalizing on innate human perceptual skills (Chang, 1990; Shu, 1988). Programming environments often employ visualization to reduce the complexity of syntactic issues such as the perplexing problem of putting semicolons at the right places in Pascal or keeping track of parentheses in Lisp.

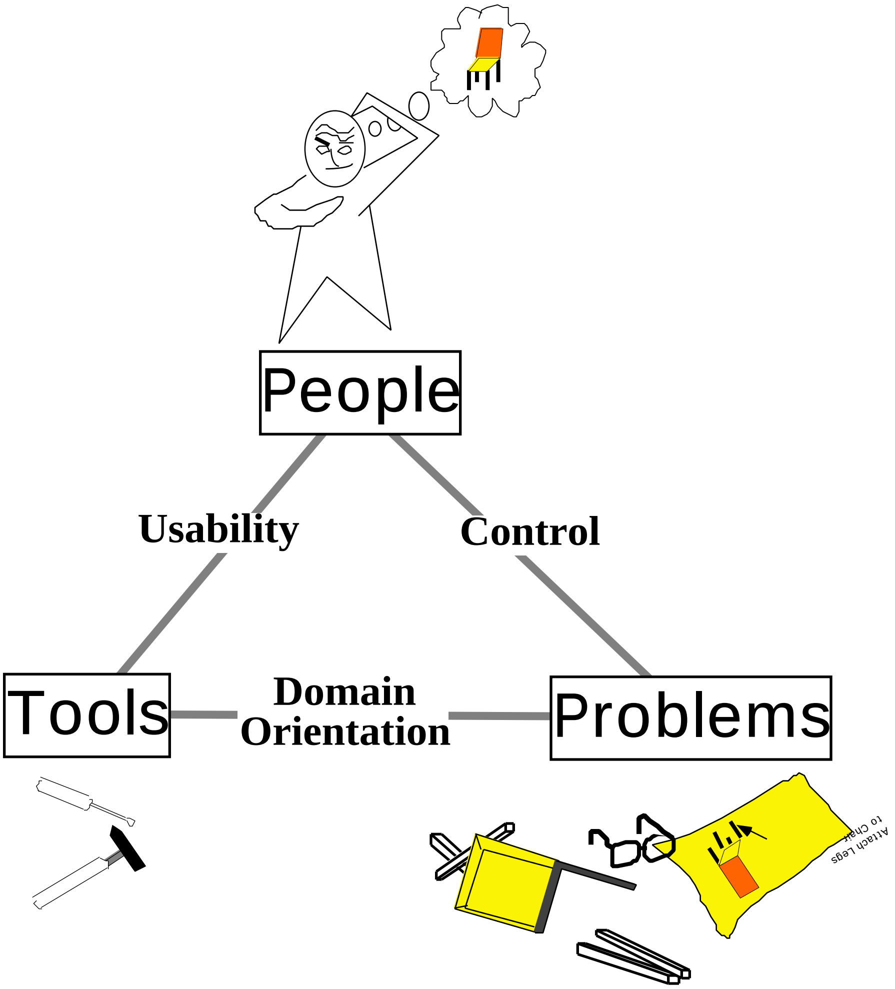


Figure 1: People, tools, and problems

Domain-orientation, in contrast to the use of visualization, is a semantic characteristic that describes the meaning of concepts without specifying their look. Programming environments with a low degree of domain-orientation, such as Pascal, Lisp, Basic, and C++, are shown in the lower half of Figure 2. These environments are unlikely candidates for end-user programming languages because they represent a big transformational gap between the problem domain and the programming domain. General purpose visual programming languages, such as Prograph (Golin, 1991), make high use of visualization.

Prograph is a programming language based on the visualization of data flow and functional programming (Figure 2, bottom right). Prograph is highly visual, and its two-dimensional syntax simplifies the task of program construction. However, the language components provided by Prograph are not domain-oriented. Visual programming has at least partially failed in its mission to lower the barriers of programming and, consequently, to enable more people to program (Brooks, 1987). To date, visualization in programming is predominately used to address syntactic issues. That is, numerous visual programming languages have emerged as new "wrappings" for classical computer science programming concepts such as iteration, sequence, and branching. Although these concepts are essential for many types of programming, the syntactic mapping of traditional text-based programming approaches to visual programming languages is questionable with respect to its effect on lowering the barriers of programming (Lewis &

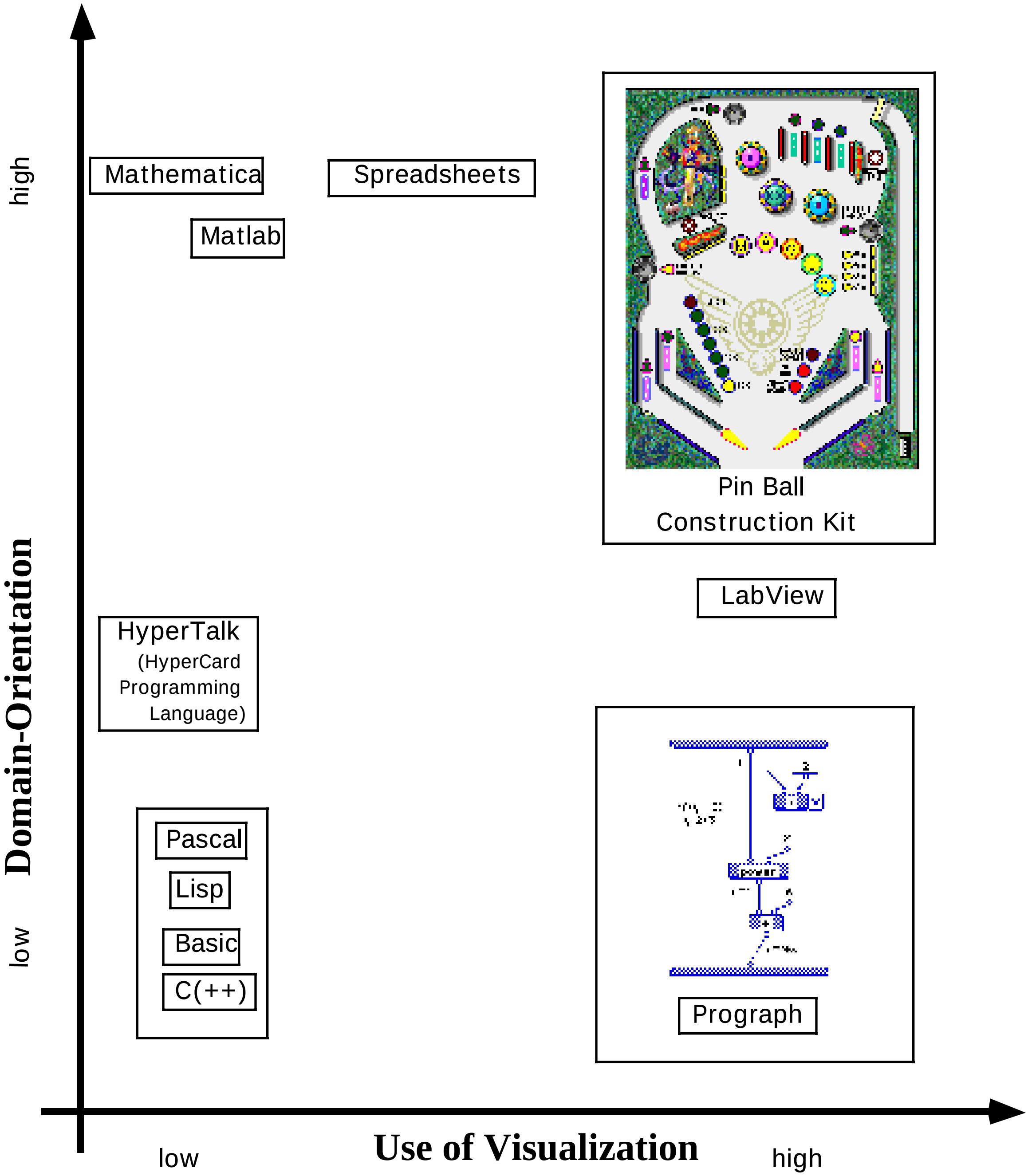


Figure 2: Languages Can Be Characterized Along the Dimensions of Visualization and Domain-Orientation

Visualization is typically used to illuminate syntactic issues in a language. Domain-orientation can be used to bridge the gap between the semantics of the problem domain and the semantics of programming concepts. The PinBall Construction Kit is a domain-oriented visual language whereas Prograph is a general-purpose visual language.

Olson, 1987). Worse yet, some studies have found that the comprehensibility of visual program representations can actually be worse than their textual counterparts (Green, Petre, & Bellamy, 1991).

Interactive programming environments such as HyperCard with HyperTalk represent a compromise between general purpose and domain-oriented languages (Nardi, 1993). Nardi points out that HyperCard-like environments are a bad compromise because they share the complexity of general purpose programming languages, such as C++, without matching their performance. At the same time, the domain-orientation of HyperCard-like environments is insufficient to enable end-users to write useful applications from scratch.

Most end-user programming languages are highly domain-oriented and, therefore, are positioned in the upper half of Figure 2. Domain-oriented languages narrow the semantic gap between the problem domain and the programming domain (Fischer & Lemke, 1988). Mathematica and Matlab feature domain-oriented operations, such as integration and Fourier transformations, that are familiar to mathematicians (Figure 2). The formula language featured by spreadsheets furnishes task-specific primitives (Nardi & Zarmer, 1993), such as taking averages, rounding numbers, and operating on dates, that are familiar to financial planners.

Domain-oriented visual programming environments elevate the task of programming to the manipulation of components directly pertinent to the tasks of end-users. In the PinBall Construction Kit (Fischer & Lemke, 1988), for instance, end-users program by manipulating bumpers, obstacles, and flippers. The combination of usability through visualization and domain-orientation makes the PinBall construction kit a useful programming mechanism for people interested in creating working PinBall applications.

2.2. Control: Degrees of Delegation

Control is an important characteristic to describe the interaction between people and problems through tools. How much of the problem-solving process can be *delegated* to other people or agents? If, for example, people are interested in solving the problem of getting bread, they have several options. Creating the bread from scratch gives people a high degree of control, as they get to select all the ingredients, but, at the same time, represents a large effort. Tasks such as kneading could be delegated to a bread baking machine. Simpler yet, people could buy bread at a bakery.

Delegation requires *autonomous* tools to reduce the problem-solving effort. Unlike passive tools, autonomous tools take the initiative to do tasks. Furthermore, autonomous tools need very little supervision to do these tasks. Consequently, autonomous tools not only allow people to interact with a problem on a more abstract level, hiding irrelevant details of the problem-solving process, but these tools help to save time. This gives people opportunities to deal with and learn about other tasks while the tools are active. In the bread baking scenario the baker working at a bakery could be considered a highly autonomous tool. A person in need of a loaf of bread does not have to instruct the baker to bake the bread not does the person have to spend any time in supervising the baker. In the following I use a theatrical metaphor (Laurel, 1993) to discuss the distribution of roles between people and tools. Specifically, I analyze these roles in human-computer interaction schemes based on *direct-manipulation* or *delegation*.

Direct manipulation provides maximal control but, at the same time, results in great effort. The computer turns into a passive tool that needs to be operated like a *hand puppet* by the user, who turns into a *puppeteer* (Figure 3). In a hypertext system, for instance, the user navigates through hyper-spaces by following links. The initiative, nonetheless, is up to users, for if they do not operate the user interface by moving the mouse or clicking buttons nothing happens.

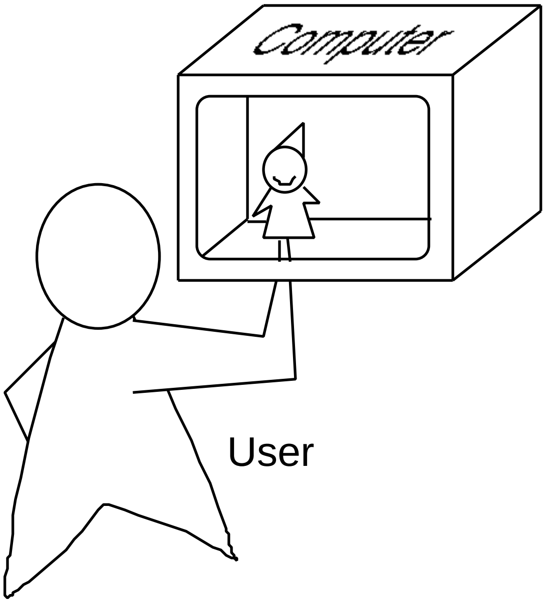


Figure 3: Direct Manipulation: Users are puppeteers

Direct manipulation can be too direct a means of control for applications that model autonomous entities such as the creatures of an artificial life world. Negroponte (1991) suggested a theatrical metaphor (Figure 4) in which tasks are delegated to actors. This approach could be considered as the opposite end of the control spectrum because once the scripts for actors have been created and the play has started, the audience (the users) is left with no control over the play. In other words, users assume the role of passive *spectators*.

Traditional simulation environments are based on delegation. The definition of simulation parameters is similar to writing the script for an actor. After the simulation has been prepared, and once the simulation has been started, the user of the simulation system becomes a spectator limited to watching the progress of the simulation.

The point here is not to advocate one approach over the other. Instead, I suggest combining the virtues of both approaches in the *participatory theater metaphor*(Figure 5) by perceiving *control as a continuous spectrum* rather than a discrete dichotomy of direct manipulation versus delegation.

Actors in the participatory theater will act according to the script unless the audience tells them to do something different. Depending on the level of participation, the interaction can assume any point in the control spectrum. If the viewers (users) do not participate then they have no control over the play and become passive observers with no effort involved. On the other hand, excessive participation will result in the user taking over the play to the level of direct manipulation.

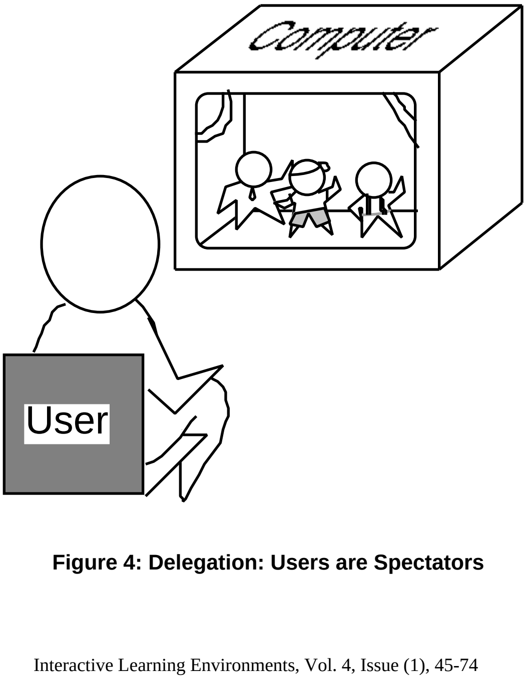


Figure 4: Delegation: Users are Spectators

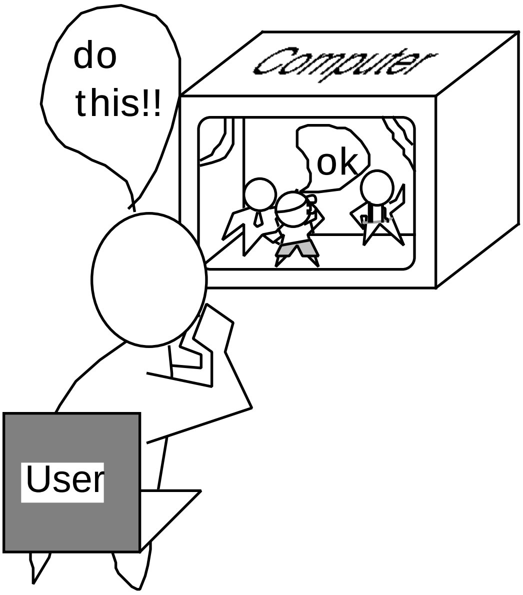


Figure 5: Participatory Theater

The point here is to note that there is no optimal way to simplify programming or baking with the “right” degree of control and delegation. The degree of delegation required to solve a problem depends on many factors including the nature of the problem, people’s skills, motivations, and the time at hand.

3. AGENTSHEETS: A PROGRAMMING SUBSTRATE TO CREATE INTERACTIVE LEARNING ENVIRONMENTS

Interactive learning environments (ILE) serve the function of tools to enable people to solve problems. In order to create learning opportunities, ILEs should address the relationships discussed among people, tools, and problems. General-purpose ILEs do not provide sufficient leverage to end-users to do interesting things in many different domains. ILEs have to be usable, be domain-oriented, and should provide end-users the desired degree of control over problem-solving processes. Instead of building ILEs from scratch, designers need to have programming substrates geared toward the creation of *effective* ILEs for end-users. This section outlines the general design principles for programming substrates, illustrates how Agentsheets addresses these principles, and describes the roles of designers and end-users interacting with Agentsheets.

3.1. Design Principles

Design principles for ILEs must address the relationship discussed above among people, tools, and problems in order to facilitate learning opportunities for designers and end-users. Programming substrates should supports these design principles.

(i) ***Usability Visual Formalisms***:: Visual formalisms are diagrammatic displays such as graphs, plots, panels, maps, outlines, and tables, with well-defined semantics for expressing relations (Nardi & Zarmer, 1993). Visual formalisms can be employed to create usable tools. Programming substrates must include visual formalisms beneficial to designers, helping them to create ILEs with effective visual representations. Visual formalisms should be versatile to allow the creation of applications in many different domains.

(ii) ***Domain-Orientation Collaborative Design***: The design of effective domain-oriented ILEs requires collaboration between designers and end-users. Programming substrates should support this collaboration. The relationship between designers and end-users of ILEs is similar to the relationship between authors and readers of books. To be most effective, designers must be aware of the end-user's needs. Programming substrates should facilitate dialog between designers and end-users. To that end, a programming substrate should serve the role of a *collaboration medium* between designers and end-users. Ideally, programming substrates endorse face-to-face collaboration sessions by including functionality for designers and end-users. These collaboration sessions are required to determine the appropriate set of domain abstractions. Collaboration is greatly enhanced if programming substrates allow fast, incremental prototyping and, therefore, encourage iterative design approaches.

(iii) ***Control Participatory Theater***: Programming substrates should support participatory theater. That is, substrates should support the design and implementation of ILEs that can adapt to end-users' needs of control over problem-solving processes. On the one hand, ILEs must enable end-users to control interesting problem-solving tasks. On the other hand, ILEs should allow end-users to delegate problem-solving tasks that are repetitive, time consuming, or simply too tedious to autonomous computational mechanism provided by programming substrates. Consequently, programming substrates must include interaction schemes combing the advantages of direct manipulation and delegation into a continuous spectrum of control.

3.2. Agentsheets: A Programming Substrate

Agentsheets (Repenning, 1991a; Repenning, 1993; Repenning & Citrin, 1993; Repenning & Sumner, 1992; Repenning & Sumner, 1994) is programming substrate for creating ILEs. In the last four years, Agentsheets has been used to create more than 40 educational and industrial applications serving as construction kits, simulation environments, visual programming languages, design environments, and games.

Similar to spreadsheets and cellular automata (Toffoli & Margolus, 1987), Agentsheets visual formalism is based on a grid structure (Repenning & Citrin, 1993). A grid location in Agentsheets may contain any number of stacked autonomous computational units called agents (Figure 6 (7)). Agentsheets agents are versatile and can be employed to create visual formalism including graphs, maps, outlines, and tables. Unlike the cells of spreadsheets and cellular automata, the look and the behavior of agents can be defined by designers. An agent may be as simple as an ordinary spreadsheet cell but it can also represent an arbitrarily complex domain-specific entity. For instance, an agent representing a car may look and act (e.g., following roads) like a car (Figure 6 (2)). All agents can:

- ***move between cells***: agents are not fixed to cells.
- ***be autonomous***: agents behave like asynchronous processes that operate concurrently with other agents or users.
- ***communicate*** : agents communicate with each other through grid coordinates or though links.
- ***have graphical depictions***: agents can have any shape and color.

- ***be animated:*** the look of agents may change over time, reflecting the state of the agent**.**
- ***play sounds:*** agents play sounds recorded by designers
- ***speak text***: agents can speak using the Macintosh speech synthesizer

The creation of effective ILEs requires tight collaboration between designers and end-users. Agentsheets supports this process with role-specific interface views. That is, end-users and designers have different tools at hand. In sections 3.3. and 3.4 an example Agentsheets application, called CityTraffic, is used to illustrate the roles of end-users and designers and the tools provided. CityTraffic helps urban planners to analyze traffic patterns.

3.3. End-users

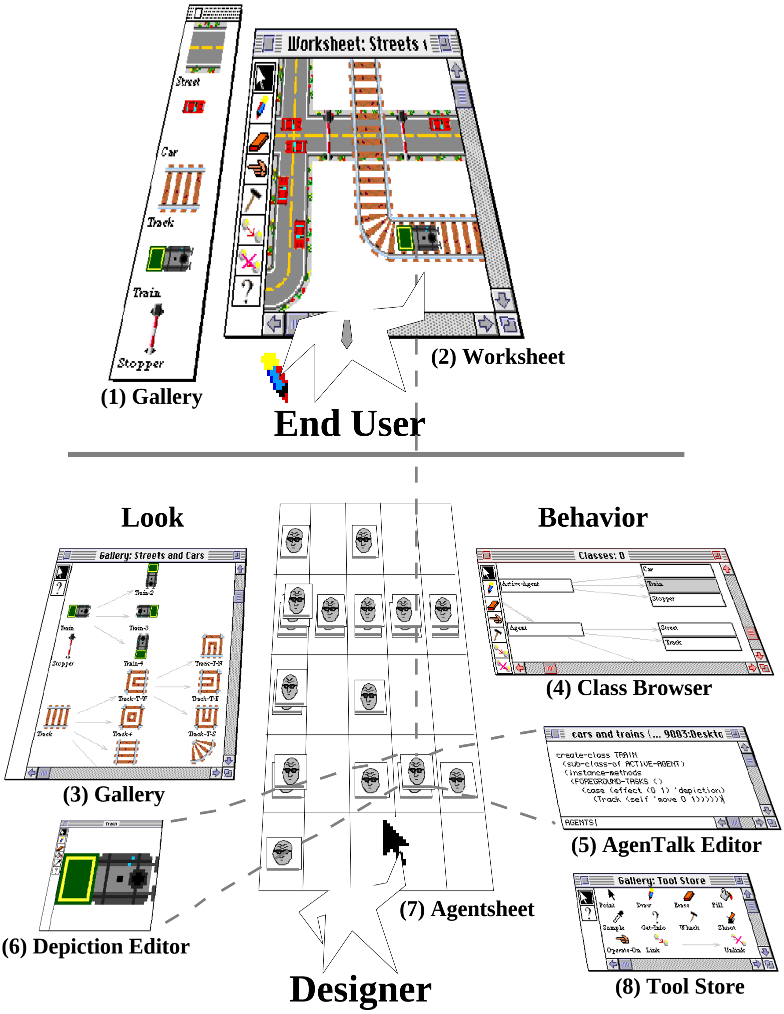
End-users create programs using language components based on familiar, visible representations pertinent to their problem domain. In the CityTraffic application (Figure 6), these components are cars, trains, streets, and railway tracks. By assembling these domain-oriented components and interacting with them end-users "program" in the sense that they define behavior. End-users have a role that is very similar to the role of a game player of SimCity. End-users:

- ***assemble components***: End-users select components in the gallery (Figure 6 (1)) and assemble them in the worksheet (2) into meaningful diagrams. For instance, they assemble individual road segments into a road system, put cars onto the roads, and install traffic signals to control traffic.
- ***interact with components***.: End-users select tools from the tool bar in the worksheet (Figure 6 (2)) and apply them to components. Tools are used to rearrange, link, and query components. In the CityTraffic application, the state and frequency of traffic lights can be changed by applying the operate-on tool, ☜, to traffic lights. Participatory theater (Repenning & Sumner, 1994) allows end-users to direct the actors (agents) without stopping the play (running application). For instance, *while cars are moving*, end-users can change parameters of cars, introduce additional traffic signals, and change the topology of streets and railway tracks. This interactive style extends direct manipulation schemes by allowing end-users to more flexibly interact with applications consisting of large numbers of autonomous agents.

3.4. Designers

In collaboration with end-users, designers create domain-oriented visual programming environments that feature components pertinent to end-users. To support the dialog between end-users and designers, Agentsheets has to provide efficient, incremental mechanisms for specifying components. Agentsheets supports designers by allowing them to:

- ***incrementally define the look of agents***: The look of an agent is defined using the depiction editor (Figure 6 (6)). In designer view, the gallery (Figure 6 (3)) is a repository of depictions that support the *incremental* creation of related depictions. New depictions can be derived from existing ones through icon transformations. In the CityTraffic application, the designer had to create all the depictions of streets, cars, railway tracks, trains, and traffic signals. The majority of icons were created automatically. For instance, the designer manually created a base depiction of a straight railway track segment, , and then automatically created variations such as bent tracks, , using provided transformations.
- ***incrementally define the behavior of agents***: Using the AgenTalk editor (Figure 6 (5)), designers define the behavior of agents. The behavior determines how agents interact with each other and how they interact with end-users through tools. The AgenTalk language (described in section 4.2) is object-oriented and, therefore, by means of inheritance, behavior can be defined incrementally. The class browser (Figure 6 (4)) helps designers locate functionality in the form of existing agent classes. By building on top of these provided agent classes, agents inherit many of their basic behaviors, such as the ability to be linked and queried. Thus, designers need only augment these inherited behaviors with behaviors specific to the problem domain.



```
create-class TRAIN
  (sub-class-of ACTIVE-AGENT)
  (instance-methods
   (FOREGROUND-TASKS ()
     (case (effect (0 1) 'depiction)
       (Track (self 'move 0 1))))))
AGENTS|
```

Figure 6. End-users and Designers

Agentsheets provides role-specific views for end-users and designers. End-users compose programs by selecting components in the gallery (1) and putting them into a worksheet (2). Designers view worksheets (2) as agentsheets (7), i.e., agents organized in a grid. They create networks of related depictions in the expanded gallery (3), design depictions with the depiction editor (6), define behavior with the AgenTalk editor (5) by reusing existing agent classes found in the class browser (4), and create or subscribe to tools in the tool store (8). Tools enable end-users to interact with agents.

In the CityTraffic application, the behaviors of cars and trains had to be defined. Cars have to follow roads, watch out for other cars, avoid collisions, and obey traffic signals.

- ***define interaction tools***: Designers provide mechanisms for end-users to interact with agents by selectively subscribing to existing tools featured in the tool store (Figure 6 (8)), extending the behavior of existing tools, or by creating new domain-specific tools. Agentsheets provides a default set of tools with predefined behaviors. For instance, applying the eraser tool, , to a car will delete that car unless the designer has specified otherwise.

4. USING AGENTSHEETS

Agentsheets includes incremental mechanisms to efficiently define the look and behavior of ILEs.

4.1. Defining the Look of Agents

Agentsheets provides tools for the efficient design of icons. This is essential to enable fast prototyping. The *gallery* (Figure 6 (1,3)) is used by a designer to create a network of related road icons. First, the designer creates a base icon (Figure 7) representing a straight street segment using the depiction editor (Figure 6 (6)). This icon will be the basis for syntactical transformations to yield variations of the street icon to represent different connectivity patterns of roads.

The transformations have emerged from analyzing related icons in earlier applications created with Agentsheets. Designers often created sets of related icons by painstakingly drawing each icon from scratch because the kind of transformations they required were not supported in traditional icon editors. Building some of the observed manual transformations by icon designers into the Agentsheets substrate has reduced the time it takes to create icon families from hours to minutes. This is especially true for sophisticated color icons.

The set of classical icon transformations found in commercial graphics applications, including rotation and flipping, has been extended in Agentsheets with more complex transformations suited for icons representing conductors of flow such as roads, railway tracks, rivers, wires, and pipes. For instance, transformations applied to the base icon (Figure 7) result in the icons shown in Figure 8. A detailed description of these transformations and their general applicability can be found in Repenning (1994).

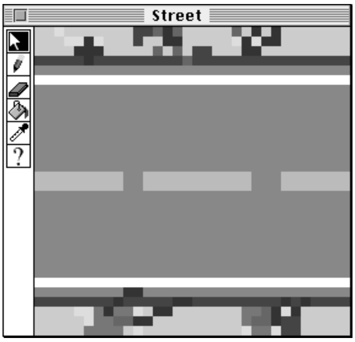


Figure 7: Base Icon representing Street

Instead of applying transformations to icons individually, designers run *transformation scripts* to create frequently used networks of related icons. In a typical scenario a designer would start by creating a set of base icons. In the gallery shown in Figure 9 the base icons for a traffic simulation have been drawn.

The designer selects the track icon and applies a transformation script to create all icons required to represent connectivity among the four adjacent neighbors of the icon. The script creates the icons and generates names for the icons (Figure 10).

Automatic transformations not only save time, they also encourage designers to experiment with different looks of icons while maintaining consistency between related icons. Without automatic transformations the need to change a base icon would force designers to manually touch up all related icons.

Transformations and transformation scripts are not hard wired. The open architecture of Agentsheets allows designers to add new icon transformations and new transformation scripts.

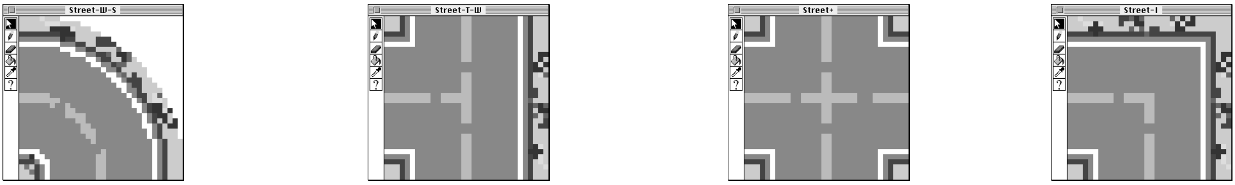
After transforming all base icons, the gallery can be used to draw a complete scene (Figure 11):

4.2. Defining the Behavior of Agents

Different programming approaches address different programming needs for designers and end-users. AgenTalk is an agent programming language typically used only by designers. Graphical rewrite rule, and programming by example are programming approaches that enable end-users to define simple behavior.

Programming Using AgenTalk

Programming using AgenTalk is the lowest level of agent programming. AgenTalk is an object-oriented extension of Lisp based on the OPUS system (Repenning, 1991b) extended with spatial primitives. Similar to *Logo (Resnik, 1992), AgenTalk is used to program concurrent computational entities (turtles and patches in *Logo and agents in AgenTalk).



Bending[1] Forking Crossing Sharp Bending

Figure 8: Transformations of Base Icon

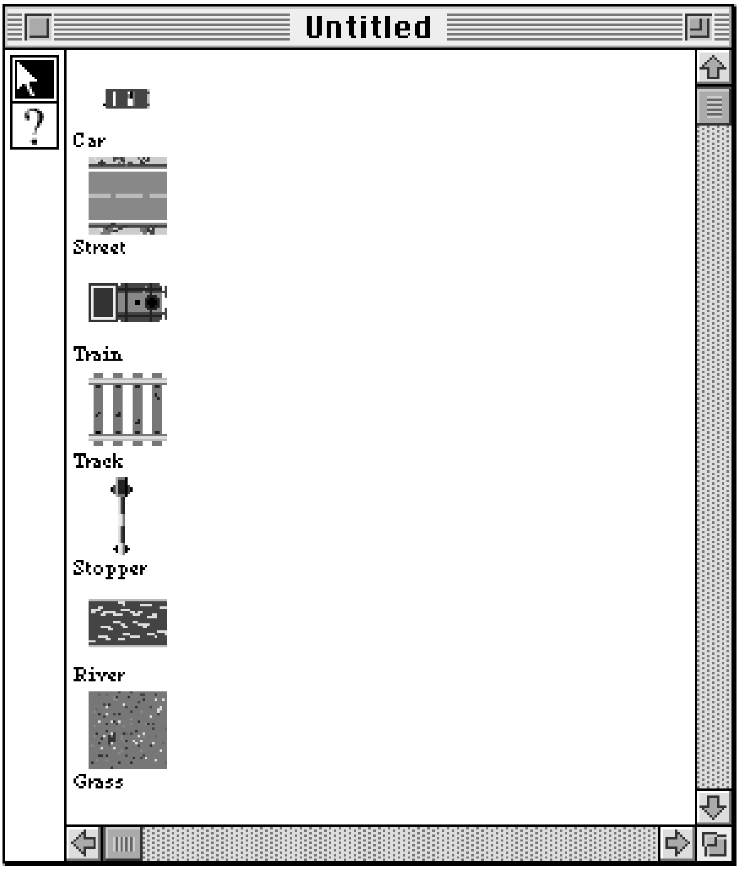
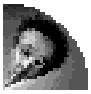


Figure 9: Gallery containing Base Icons

Unlike *Logo, however, AgenTalk is not intended to be an end-user programming language but, instead, is geared more toward designers who are experienced programmers. To that end, AgenTalk includes object-oriented mechanisms such as class inheritance to build a large number of heterogeneous agent classes from existing ones. In order to enable participatory theater, AgenTalk includes primitives to simultaneously deal with direct manipulation and delegation (Repenning, 1993). That is, designers use AgenTalk to define how agents react to mouse input and to define what agents should do autonomously. Due to the general-purpose nature of the underlying

[1] Base icons can be arbitrary; for instance bending icon will create icon .

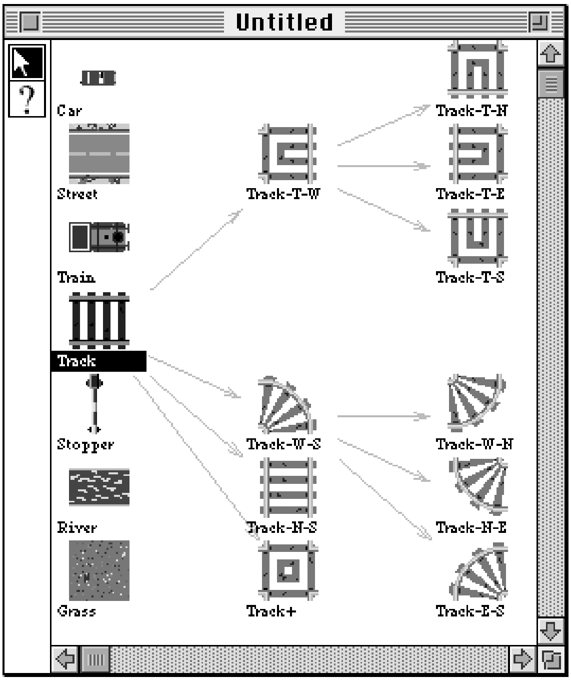


Figure 10: Transformed Tracks

implementation language, Common Lisp, AgenTalk is the most flexible way to program, but it is also the most demanding approach to define behavior.

The programming task is to define the behavior of a train, , to follow railway tracks, :

```
(1)  (create-class TRAIN
(2)    (sub-class-of ACTIVE-AGENT)
(3)    (instance-methods
(4)      (FOREGROUND-TASKS ()
(5)        (case (effect (0 1) 'depiction)
(6)          (TRACK (self 'move 0 1))))))
```

The TRAIN class (1) is defined to be a subclass of ACTIVE-AGENT. Active agents are autonomous agents receiving FOREGROUND-TASKS messages to initiate actions. The case statement (5) contains a spatial operator, (effect (0 1) 'depiction), returning the name of the depiction to the right of the train agent. If the depiction of that agent on the right is a TRACK[2] depiction (6) then the train moves[3] to the right. As soon as the class is defined, and the agent scheduler is activated, the train begins to follow the tracks. The Agentsheets scheduler (Repenning, 1993) enables users, at any point in time, to interact with the train by using the mouse to move it to some new position or by applying a tool to the train.

Programming on the AgenTalk level is demanding and is typically done only by designers and not by end-users. AgenTalk programming requires at least a minimal knowledge of Lisp and familiarity with object-oriented principles.

Programming with Graphical Rewrite Rules

The graphical rewrite rule approach of programming allows the definition of simple behavior by manipulating pictures. Rewrite rules have been explored by Lieberman (1987) in the Tinker system, by Furnas (1991) in the BitPict system, and by Bell (Bell 1991; Bell, 1992; Bell, Rieman, & Lewis, 1991) in ChemTrains. The

[2] The association between an icon and its name to be used in AgenTalk code is established in the gallery by naming the icon.

[3] On a MacIIfx a train can move at a rate of about 120 track units per second

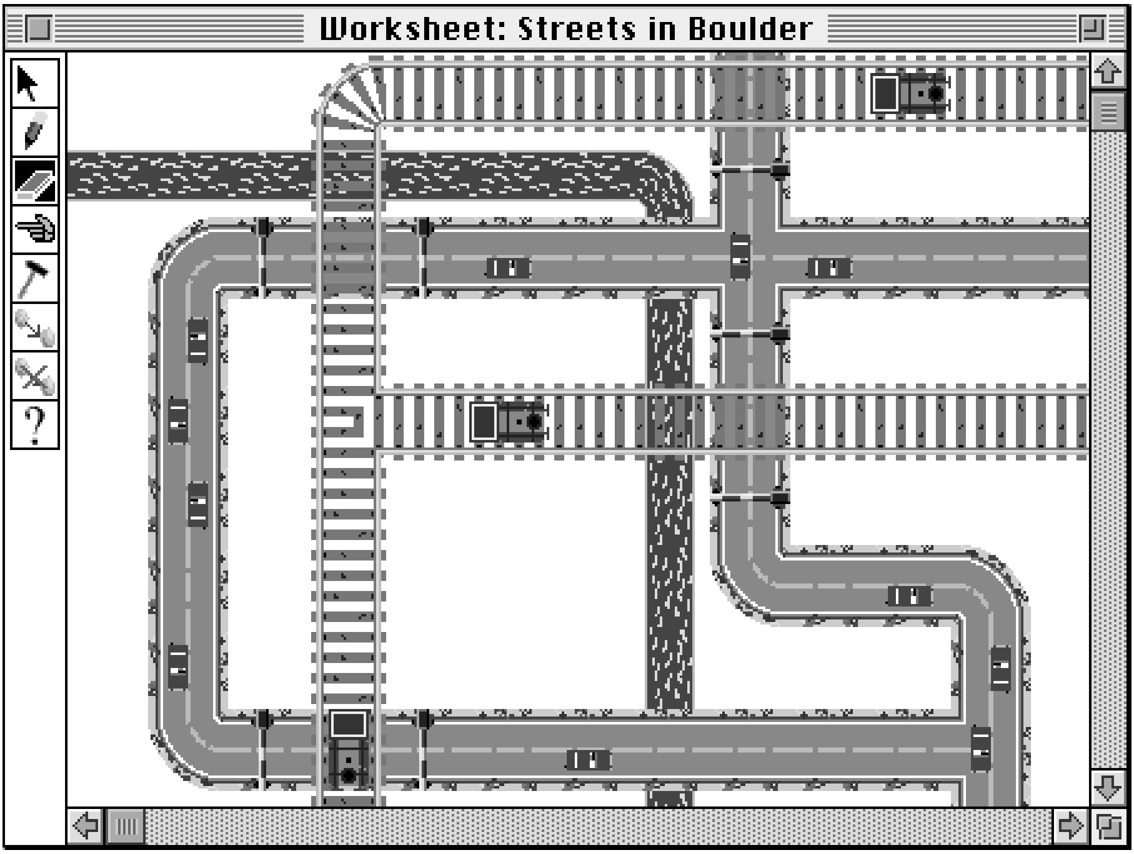


Figure 11: CityTraffic Simulation for Urban Planners

BitPict system is limited to graphically reason about pixels. ChemTrains can deal with more complex objects such as boxes, circles, and entire bitmaps. Both BitPict and ChemTrains have no included abilities to augment graphical rules with textual predicates. The Vampire system overcomes this limitation with attributed graphical rules (McIntyre & Glinert, 1992).

Graphical rewrite rules trade opportunities of learning about programming for learning through programming. End-users can employ graphical rewrite rules to express simple agent behaviors without having to understand difficult programming concepts such as multiple inheritance from object-oriented programming. Concepts relevant to a problem domain such as movement can be easily represented with graphical rewrite

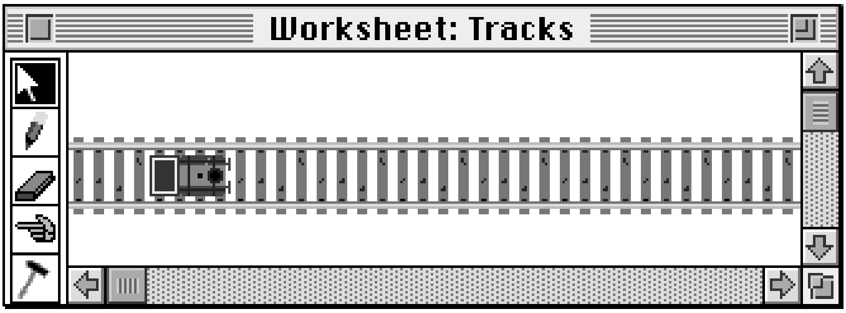


Figure 12: City Traffic Scene

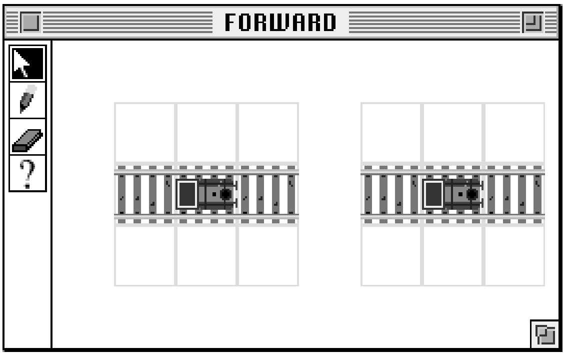


Figure 13: Rule Editor

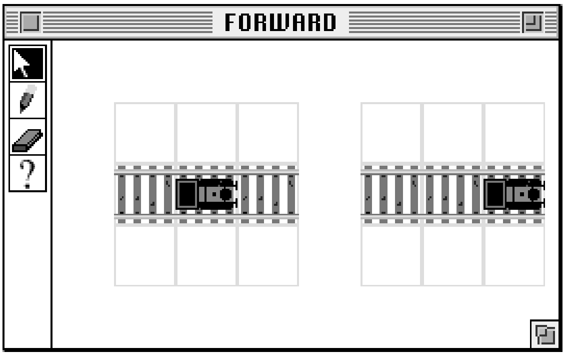


Figure 14: Train is selected and moved to right

rules.

To program a train using graphical rewrite rules an end-user would start by drawing a scene containing relevant agents (Figure 12):

By double-clicking the train, and after defining the name of the rule to be "forward" the rule editor pops up (Figure 13):

The left hand side of the rule, the IF part, shows the situation the train was in at the time of creating the rule. The right hand side of the rule, the THEN part, shows the future state of the train. Initially the THEN part is identical to the IF part. The user programs the train by selecting the train in the THEN part and dragging it one position to the right (Figure 14).

In complex rules the relationship between agents in the IF part and the THEN part can become confusing because it is unclear what agents in the IF part correspond to what agents in the THEN part. Correspondence is crucial if the elements in a rewrite rule are not just pictures but also represent complex hidden states. The *correspondence problem* of the graphical rewrite rule is solved in Agentsheets using a technique borrowed from data analysis called *brushing*. If an agent exists on both sides, then selecting it will also select its corresponding agent. Selecting the train in the THEN part and moving it also selects the corresponding train in the IF part. Agents that exist only in the THEN part are agents that will be created; agents that exist only in the IF part will be erased.

The created rule is activated when the rule editor is closed. Immediately the train begins to move until it reaches the end of the tracks (Figure 15).

Rules can be generalized or specialized. Adding agents to the IF part of a rule will make the rule more specific. Removing agents from the IF part, on the other hand, will generalize the rule. For instance, a more general "forward" rule eliminates the tracks on the left and right of the train (Figure 16):

A big problem of rewrite rules is *spatial relation literalism*. That is, rewrite rules tend to describe overly specific situations in terms of where things are in a scene and what orientation

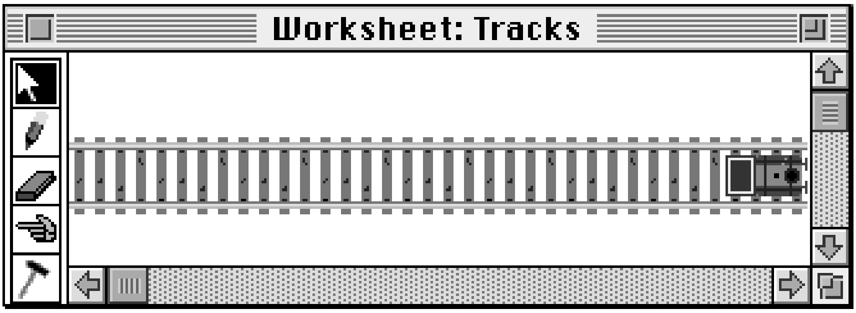


Figure 15: Train has moved to end of track

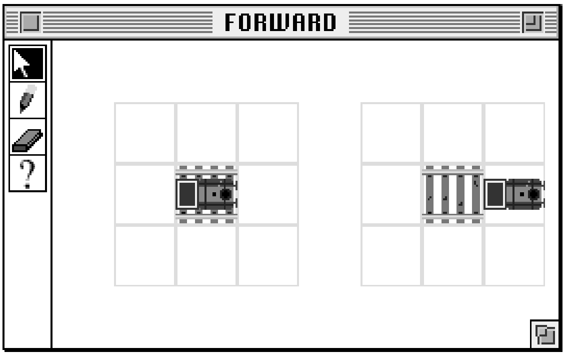


Figure 16: Generalized “Forward” Train Rule

they have. For instance, in KidSim (Smith, Cypher, & Spohrer, 1994) and in early Agentsheets (Repenning, 1993) the rule in Figure 16 has a very rigid Euclidean interpretation: If the train is on top of the track shown then the train can move to the right. Yet, most Agentsheets users when shown this rule would expect the train to be able to follow a complex system of tracks including turns, forks, and intersections such as the one shown in Figure 11. In order to enable the train to master complex systems of tracks with the literal graphical rewrite rules, a user would have to define 44 different rules (11 track variants from Figure 10 times 4 train variants corresponding to four possible train directions). Instead, Agentsheets features combined *Euclidean/topological rewrite rules*. Users can attach topological semantics to icons in the gallery. The semantics attached to icons get transformed in addition to the look of the icons (Repenning, 1994). The graphical rewrite rule interpreter takes this additional information from the gallery into account. Consequently, the number of rules required to define the “train follows any track” behavior drops from an overwhelming 44 to 1. In essence, the information stored in the Agentsheets gallery about related icons is employed by the graphical rule interpreter to make better informed inferences.

BitPict, ChemTrains, and Vampire employ, in contrast to Agentsheets, the notion of a global rule set. A centralized algorithm matches the rules in the rule set with the current picture. If the left-hand side of a rule is satisfied then the rule will fire by executing the right-hand side of the rule. The graphical rewrite rule approach used in Agentsheets makes use of decentralized rule sets. That is, each agent has its own set of rules. This localistic rule matching has been used in Agentsheets to keep state and function defined by rules together. This is helpful because rule-based agents preserve their object-oriented nature. Analogous to any other agent, the behavior of rule-based agents can be refined via class inheritance.

Programming by Example

The programming-by-example approach (Cypher, 1993; Kurlander & Feiner, 1992; Myers, 1988) employed by Agentsheets (Figure 17) can be viewed as an extension of the graphical rewrite rule approach. A program acquisition agent observes the user modifying artifacts and, similar to the agent in Object Lens (Lai, Malone, & Yu, 1989), creates a program for the user. The program contains spatial operations such as moving, deleting, or adding an agent as well as non-spatial operations such as querying the attribute of some agent. User operations are interpreted by the program acquisition agent to create programs consisting of IF-THEN rules that get attached to the agents to be programmed.

In programming by example, the program arises from the interaction among users, artifacts to be programmed (Agentsheets worksheets), program acquisition agents, and rules. Unlike programming approaches based on the detection of repetitive user actions, such as Eager (Cypher, 1993), the programming by example mechanism employed in Agentsheets is able to create programs based on a single sequence of user operations.

Program acquisition agents observe the interactions between users and artifacts. Based on their observations program acquisition agents create and modify rules. Two types of user-artifact interactions are distinguished:

- ***Queries***: Queries do not change the artifact in any way. For instance, users can query the value of agent variables without changing them.
- ***Modifications***: Modifications change artifacts. Changing the value of an agent variable, changing the look of an agent, or moving an agent in the worksheet are examples of modifications.

Based on the type of user-artifact interaction program, acquisition agents make different suggestions. Queries lead to *justifications*. For instance, if a user inspects the value of an agent variable the program acquisition agent assumes that the value of that variable is important to the user and, therefore, suggests the value of the variable as justification. Modifications, on the other hand, are

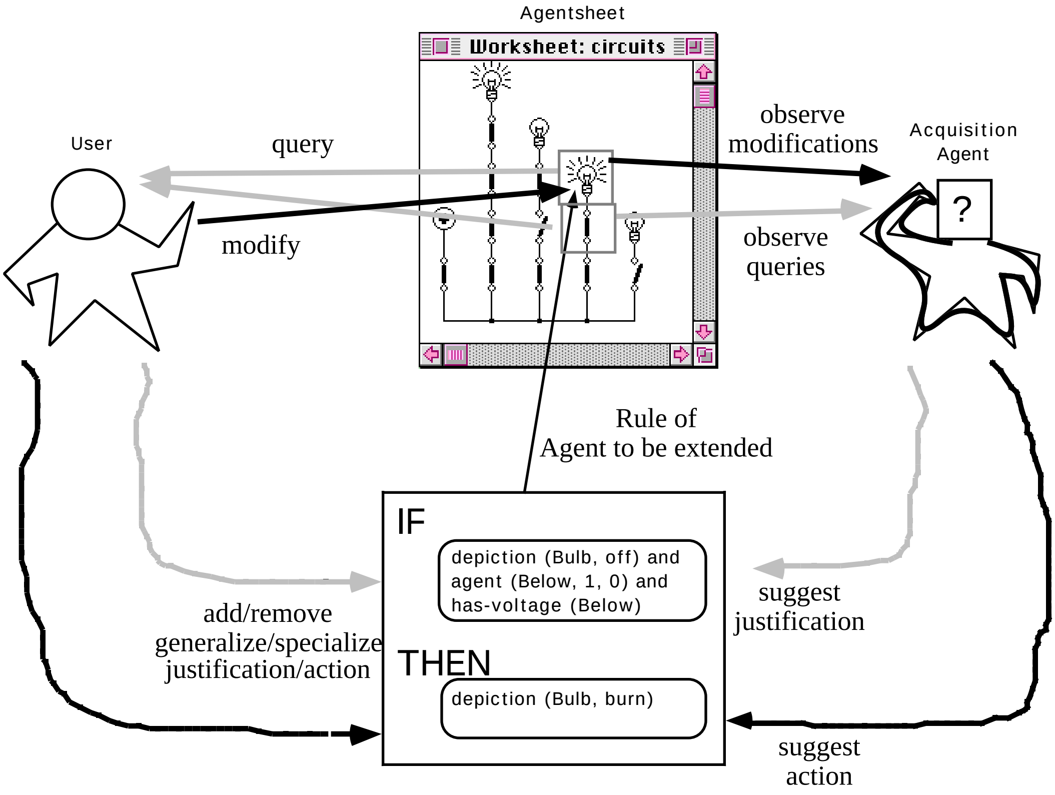


Figure 17: Interaction between User and Program Acquisition Agent

interpreted by the program acquisition agent as *actions*. If, for instance, a user changes the look of an agent then the program acquisition agent will record a change-of-look action.

Users can edit rules. Justifications or actions suggested by the program acquisition agent are irrelevant if they are due to unnecessary or even erroneous user operations. Users can simply delete justifications or actions. In more complex cases often the action or the justification suggested by the program acquisition agent is too specific. Since it is hard to infer abstract concepts from concrete manipulations the program acquisition agents have been provided with a repertoire of simple spatial generalization rules. By pressing a "generalize" button, users ask program acquisition agents to offer a menu of generalizations. For instance, a justification such as "because A is immediately above B" can be generalized to "because A and B are in the same column," "because A is at distance 1 to B," or "because there is an A and a B." If the user is not satisfied with any of the generalization options offered by the program acquisition agent then the user can manually edit expressions.

Users interact with program acquisition agents (Figure 18). Program acquisition agents can be created, selected and personalized by users. The program acquisition agent that is selected is in charge of making suggestions for justification and actions.

Based on their user modifiable profiles, program acquisition agents can detect different user operations and react differently. For instance, the "Tidy" program acquisition agent is personalized by the user to be very perceptive (Figure 19). If

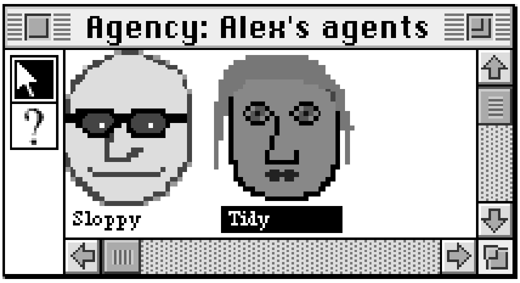
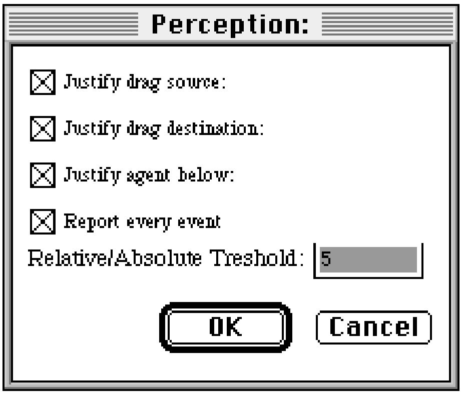


Figure 18: An Agency with two Program Acquisition Agents



Perception:

☒ Justify drag source:

☒ Justify drag destination:

☒ Justify agent below:

☒ Report every event

Relative/Absolute Treshold: 5

Figure 19: Perception Parameters of Program Acquisition Agent “Tidy”

users drag an agent in a worksheet, Tidy will use the agents located at the source and the destination of the drag operation to be justifications.

To program a train using programming by example in Agentsheets an end-user would start by drawing a scene containing relevant agents (Figure 20).

By double-clicking the train, Sloppy has been selected in the Agency window (Figure 18). After defining the name of the rule, an empty programming-by-example rule editor appears on the screen (Figure 21):

The Watch User flag of the rule editor is enabled. Consequently, Sloppy, the program acquisition agent, will observe all the operations of the user. First, the user selects the railway track in front of the train (Figure 22).

Sloppy notes the selection. Because the selection does not change the worksheet in any way, Sloppy suggests the presence of the railway track as a justification for the Forward rule (Figure 23).

Sloppy uses relative coordinates in the justification because the distance between the agent to be programmed and the agent selected is below the relative/absolute threshold defined in Sloppy’s and Tidy’s profile (Figure 19). Next, the user selects the train and moves it to the right (Figure 24).

Sloppy notes the selection of the train and suggests the presence of the train as another justification of the rule. The movement of the train changes the worksheet and, therefore, is perceived by Sloppy as action (Figure 25).

Closing the rule window will assert the rule to the train agent. The programming-by-example episode is complete. The train applies the Forward rule several times and moves to the end of the railway track (Figure 26).

5. EXPERIENCE WITH AGENTSHEETS

This chapter is about the experience gained from people using Agentsheets. It describes a number of ILEs created with Agentsheets. For each ILE some learning opportunities are listed. The learning opportunities are only examples of the ones that we observed. They are opportunities and, therefore, there is no guarantee that other students would learn the same things about a particular ILE.

Some of the ILEs were created by students (exposed to C) who in the beginning were less than enthusiastic about the Lisp-like AgenTalk programming language. However, in the end, they wrote thousands of lines of AgenTalk code without being coerced to do so. We believe this attitudinal change was due to the incremental nature of Agentsheets and the intrinsic visualization of agents.

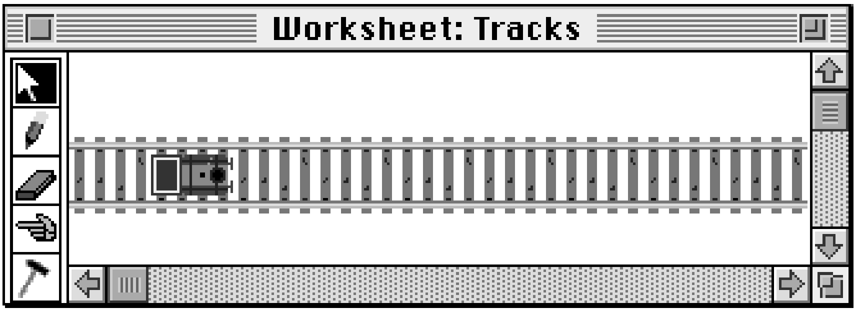


Figure 20: Initial Scene of Train on Track

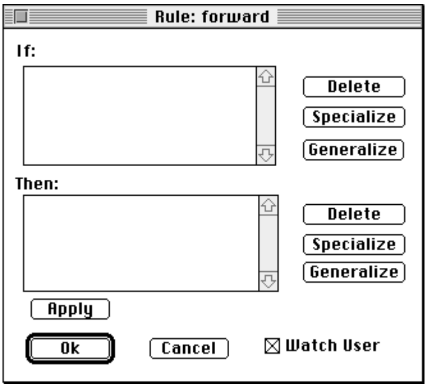


Figure 21: Programming by Example Rule Editor

5.1. CityTraffic

CityTraffic, used throughout this paper as a programming example, is a traffic analysis ILE created by environmental design students to experiment with different road topologies and to study the impact of traffic signs on the flow of traffic. Cars, , move on roads, , and hopefully observe traffic signs, , , . The goal of users is to create efficient traffic systems with a maximum flow of cars but minimal chance of traffic accidents (Figure 27).

Participatory theater allows users to interact with running simulations. That is, while the simulation is running, cars can be added, the topology of the road can be changed, or traffic signals can be installed. Learning opportunities:

- ***Learning through Programming.*** Causality: what is the relationship between trains and tracks; does the track cause the train to move in restricted ways or is the train the active part taking tracks into consideration for movement?
- ***Learning about Programming.*** Ontology: what kind of class hierarchy makes sense to appropriately represent the static and dynamic relationships among cars, trains, roads, and track? Exception handling: what should happen to cars and trains if they end up on a patch of grass?
- ***Learning by Using.*** How many cars are necessary to justify the need for increased signals? What are the trade-offs between yield signs and traffic lights?

5.2. Electric World

The Electric World application features two types of flow: the flow of electricity and the flow of magnetic fields. Both flow types coexist. An electric coil, , emits an electromagnetic field if current is flowing through the coil. The coil and the bulbs, , are implicitly grounded. A switch sensitive to electromagnetic fields is located on the left of the coil. The combination of coil and electromagnetic switch results in a solenoid (Figure 28). Learning opportunities:

- ***Learning through Programming.*** Models of flow: what happen to flow if conductors fork or join; should electricity be conceptualized as a liquid or as a set of discrete particles?
- ***Learning about Programming.*** Reuse: how can the large number of wire types (16 combinations resulting from inputs/outputs from 4 different directions) be mapped to a small number of generic but easy to understand classes?
- ***Learning by Using***. Feedback: a circuit such as the one shown in Figure 28 can lead to feedback; current causes the coil to

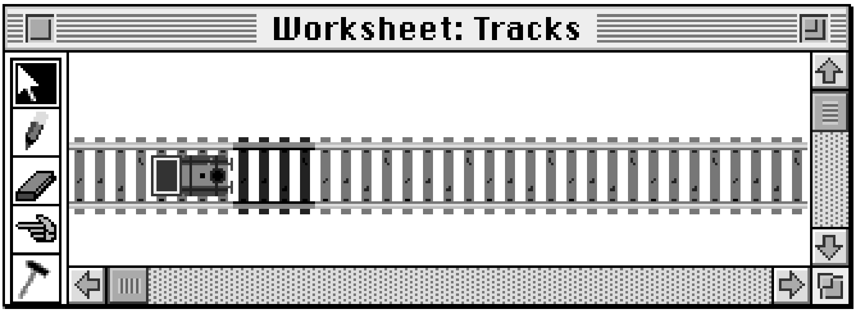


Figure 22: Railway Track in front of Train is selected

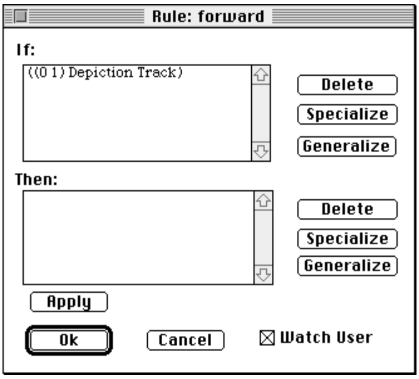


Figure 23: Rule after Selecting Track

- generate a magnetic field that via an electromagnetic switch interrupts the current to the coil.

5.3. Petri Nets

Petri Nets are used to model parallel processes. Tokens, ●, reside in places, ○. Places are connected to each other through links. The tokens flow from one place to another place through transitions, ▭, that must be fired. Tokens, places, and transitions are agents. The Petri Net application was created in one afternoon by someone interested in experiencing the dynamic aspects of Petri Nets (Figure 29). Learning opportunities:

- ***Learning through Programming.*** Theoretical models: how do PetriNets work; what are they good for; how can deadlocks be detected?
- ***Learning about Programming.*** Concurrency: what are the issues to be considered when dealing with parallel execution of code to drive the simulation?
- ***Learning by Using.*** It is very easy to create a Petri Net leading to a deadlock.

5.4. Tax The Farmer

How should a farmer be taxed for water pollution on the basis of land topology? Pollutants of farming land are washed into nearby rivers by rain. The amount of pollution washed into the river depends on the land topology. This amount needs to be determined in order to tax the farmer accordingly and to predict the needs for treatment plants (Figure 30).

This project started as a wood model. Very quickly, it became apparent how tedious it was for users of the model to manually simulate large numbers of rain drops washed into rivers. The Agentsheets solution modeled raindrops with autonomous agents that, according to the soil topology, would move themselves toward the water and compute the amount of poison accumulated. Learning opportunities:

- ***Learning through Programming.*** Causality: what is the relationship between poisoned patches of land and raindrops; are the raindrops extracting poison from patches or are patches forcing poison onto raindrops?
- ***Learning about Programming.*** How should the altitude of land patches be modeled?
- ***Learning by Using.*** The altitude model implemented turned out be problematic because it allowed the creation of physically impossible topologies (corresponding to Escher's staircase).

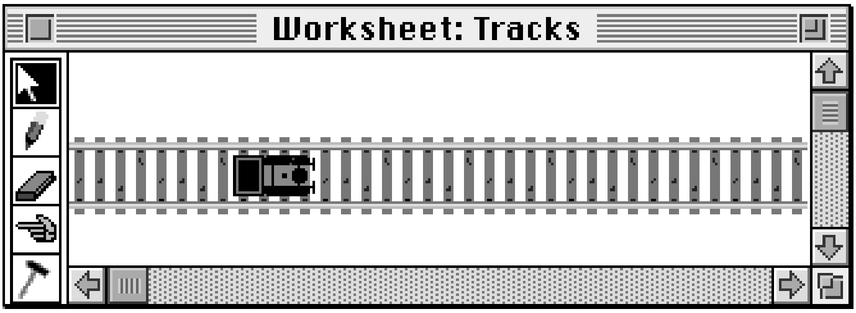
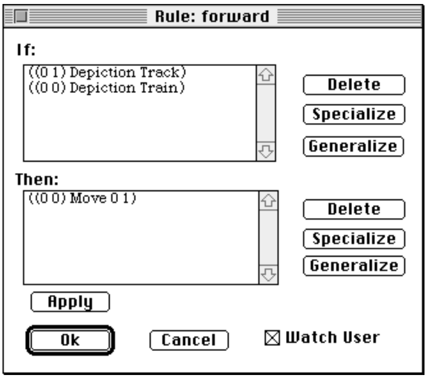


Figure 24: Train is selected and move one position to the right



((0 1) Depiction Track)
((0 0) Depiction Train)

((0 0) Move 0 1)

Figure 25: Rule with two Justifications and one Action

5.5. Particle World

In this naive physics model movable ● and fixed ▩ particles interact with each other through collisions (Figure 31). Students can draw scenes consisting of particles and experience physical as well as statistical phenomena emerging from simple behavior. Learning opportunities:

- ***Learning through Programming***. Physics: under what circumstances can fixed particles prevent the free fall of movable particles?
- ***Learning about Programming.*** Non-determinism: how should events with random components be implemented?
- ***Learning by Using.*** Distribution: certain combinations of equal distributions can lead to a normal distribution. Particles not constrained horizontally pile up to pyramids.

5.6. Rocky's Other Boot

Rocky's Other Boot is an educational environment for students to learn about causality and digital circuits. A teacher gives the students tasks to design circuits that detect a combination of spatial features of targets. For instance, a task could be to build a circuit that detects a target containing a cross or a circle and a triangle (Figure 32). Students assemble circuits by selecting individual gates and wire them up.

Rocky's Other Boot was inspired by the original system called Rocky's Boot. Rocky's Other Boot helps students to debug circuits using a *localistic contextualized explanation* facility. Every agent that represents a gate can explain itself using voice output based on input and state. Learning opportunities:

- ***Learning through Programming.*** Logic: how does Boolean logic work and how can it be used to implement a circuit detecting a particular set of patterns?
- ***Learning about Programming.*** Concurrency: how can gates and wires be programmed to act in parallel?
- ***Learning by Using.*** Real time issues: the voice explanation travels trough the entire circuit the same way as the electrical signal. This explanation reveals real-time problems such as signal racing.

5.7. Voice Dialog Design Environment

The Voice Dialog Design Environment (Repenning & Sumner, 1992) is an industrial application used to design complex phone-based user interfaces (Figure 33). Designers and customers of U S West can use this visual language to quickly prototype the very constrained

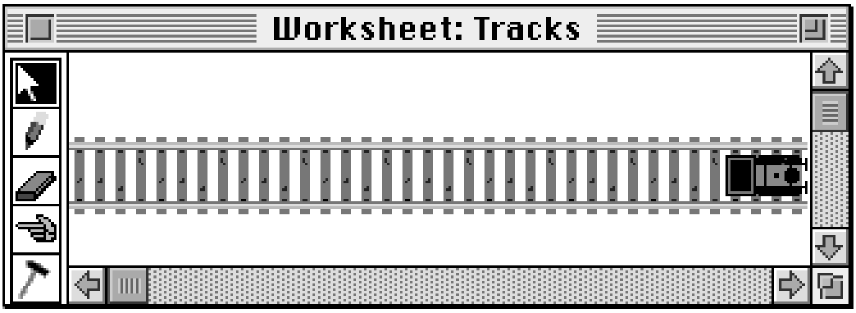


Figure 26: Train has moved to the end of the Railway Tracks

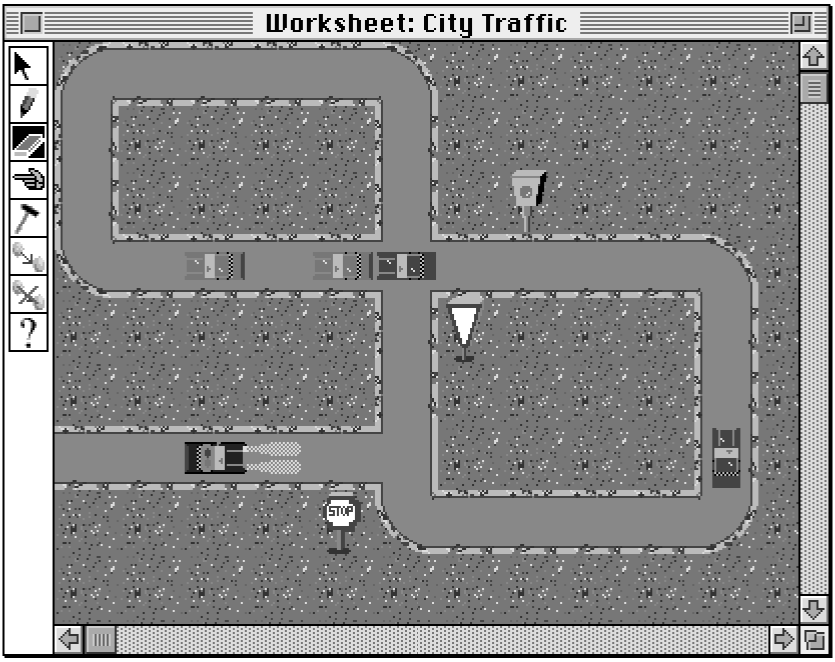


Figure 27: City Traffic

interaction between people and information services through phones. The visual language includes speech output and touch-tone button input.

Voice dialog applications are a relatively new design domain. Typical applications include voice mail systems, voice information systems, and touch-tone telephones as interfaces to hardware. Involvement in this field was part of a collaborative research effort between the University of Colorado and U S West's Advanced Technologies Division.

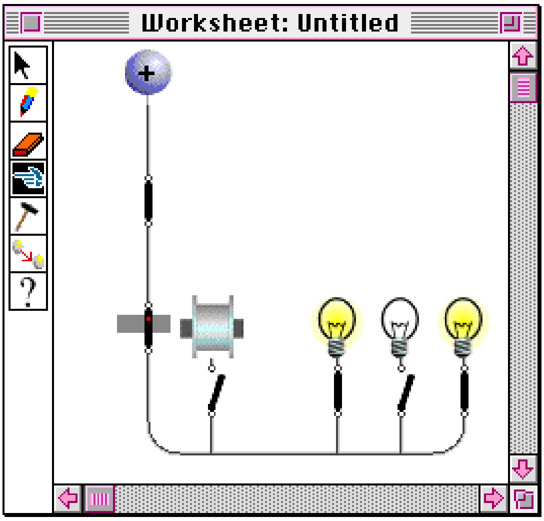


Figure 28: Electric World

Learning opportunities:

- ***Learning through Programming.*** Design: what are good representations of voice dialogs helping U S West designers to efficiently create useful designs for customers?
- ***Learning about Programming.*** Control flow: how can control flow representing a voice dialog be represented effectively on a two-dimensional display?
- ***Learning by Using.*** User interface design: what design principles hold for good voice dialogs? For instance, what is a good upper number of voice menu options?

5.8. Party Planner

The Party Planner, inspired by Rich Gold (Dewdney, 1990), deals with the optimal arrangement of people (or agents) at a party. Every agent at the party likes the other agents at the party to some degree, captured by the so-called social distance. According to behavioral scientists, social distance is an indicator for the ideal Euclidean distance between people. The need to be close to somebody reflects liking somebody.

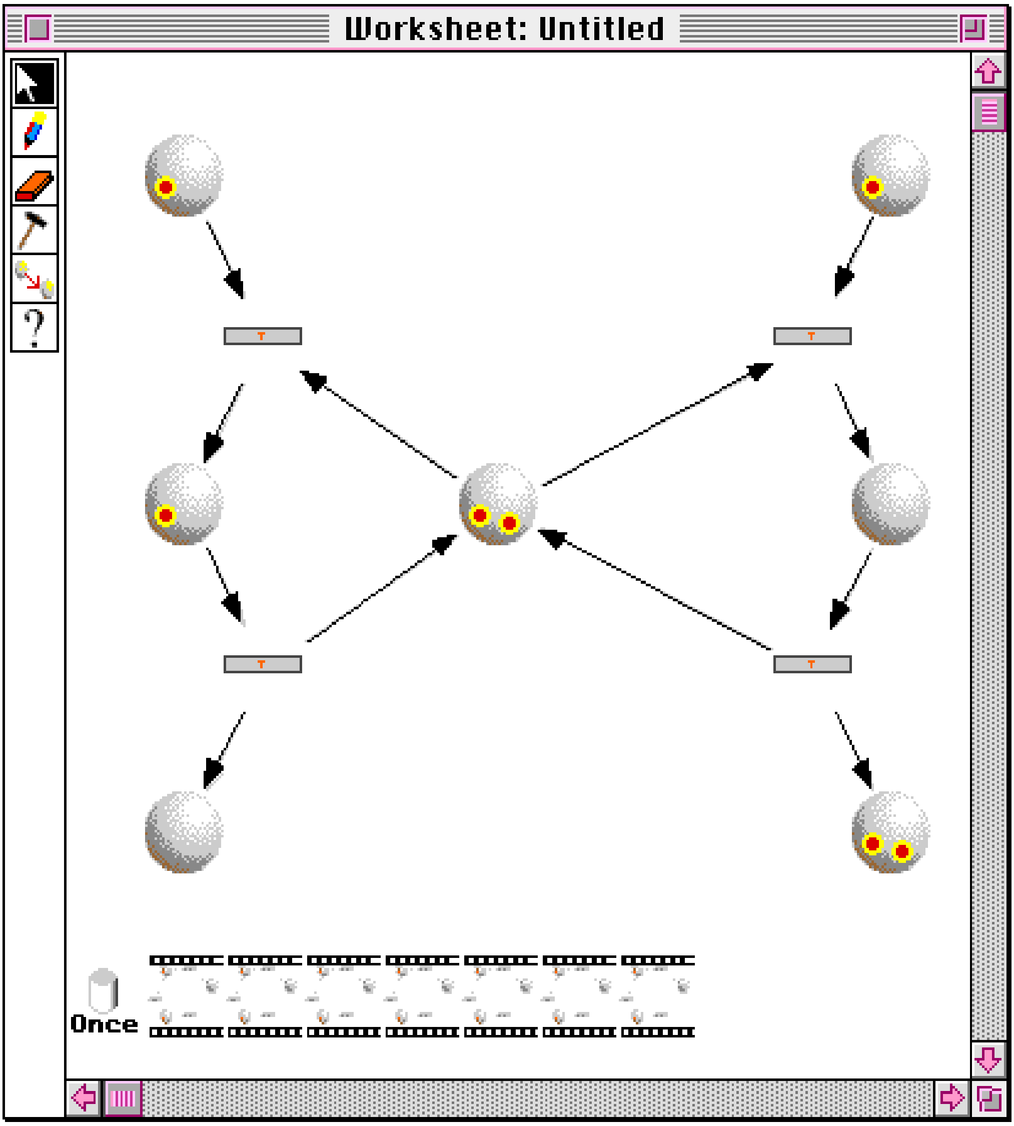


Figure 29: Petri Net modeling two Processes sharing one Resource

Every agent at any point in time is trying to optimize its happiness by being as close to the other agents of interest as implied by the social distance; close to the liked agents and far away from the disliked agents. One of the dilemmas arising from this process include that moving close to some liked agent may increase the distance to some other liked agent or decreases the distance to some other disliked agent.

The situation depicted in Figure 34 contains 4 agents. Three agents are involved in a complex triangular relationship: A: , B: and C: , where A likes B but B hates A, B likes C but C hates B, and C likes A but A hates C. None of the agents involved in this relationship will ever become completely happy because the agents' interests are not mutual. Consequently, a very complex dynamic behavior will arise, leading to a wild chase of agents. A fourth agent, called the "party pooper" , is introduced. The pooper likes everybody else but, at the same time, is hated by everybody.

Unlike in the original Party Planner by Gold, described by Dewdney (1990), the users can participate in the party theater.

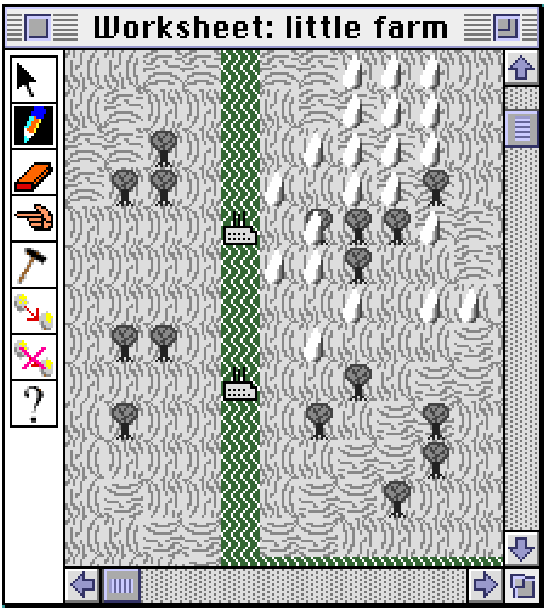


Figure 30: Farming Land with Raindrops and River

For instance, the chase, resulting from the relationships among A, B, and C, can be influenced by the user by moving agents to new locations, changing social distances, locking agents up (by building walls around them), or introducing new agents.

Learning opportunities:

- ***Learning through Programming.*** Dynamic optimization: concurrent hill climbing is a way to conceptualize many different problems involving autonomous units.
- ***Learning about Programming.*** What are efficient means of communication in situations where everybody needs to be able to communicate with everybody else?
- ***Learning by Using.*** Many asymmetric relationships have no static solution but lead to interesting dynamic patterns. The principles involved in party planning also describe the way people spread out in an elevator. The same principles could be applied to determine where to plant families of trees.

5.9. Kitchen Planner

The hill-climbing metaphor, as it is employed in the Party Planner, can be used as a constructive design force. In the Kitchen Planner, appliances such as sinks, refrigerators, and ovens are active components that try to maximize their happiness (Figure 35). Design knowledge provided by users in the form of spatial relationships that reflect kitchen design guidelines gets attached to components. Unlike in critiquing systems such as Janus (Fischer, Lemke, Mastaglio, & Morch, 1991), the knowledge is *constructive* and not just *evaluative*. That is, guidelines can be used not only to critique an existing situation but, additionally, they can suggest improvements.

The kitchen design space is conveyed to the user through tactile experience. That is, users experience relationships between design components, such as the work triangle, by “touching” and moving components. Tactile interaction makes use of principles intrinsic to the participatory theater metaphor. On the one hand, the user can express design intentions through direct manipulation by simply moving components. On the other hand, components are autonomous in trying to optimize their happiness according to the guidelines attached to them. This can lead to conflicts between the intentions of the user and design guidelines as well as to conflicts between guidelines. To that end, users can modify the strengths of guidelines and can freeze positions of components.

Learning opportunities:

- ***Learning through Programming.*** Design spaces can be conveyed through tactile experience.
- ***Learning about Programming.*** Who owns constraints involving multiple design units?
- ***Learning by Using.*** Some appliances need to be frozen in space; otherwise, the kitchen cannot be controlled any more by users.

5.10. Pack Agent

Pack Agent is an educational game environment used to “lure” students into programming. Arcade games, such as Pack Agent, are instances of participatory theater. The objective of the

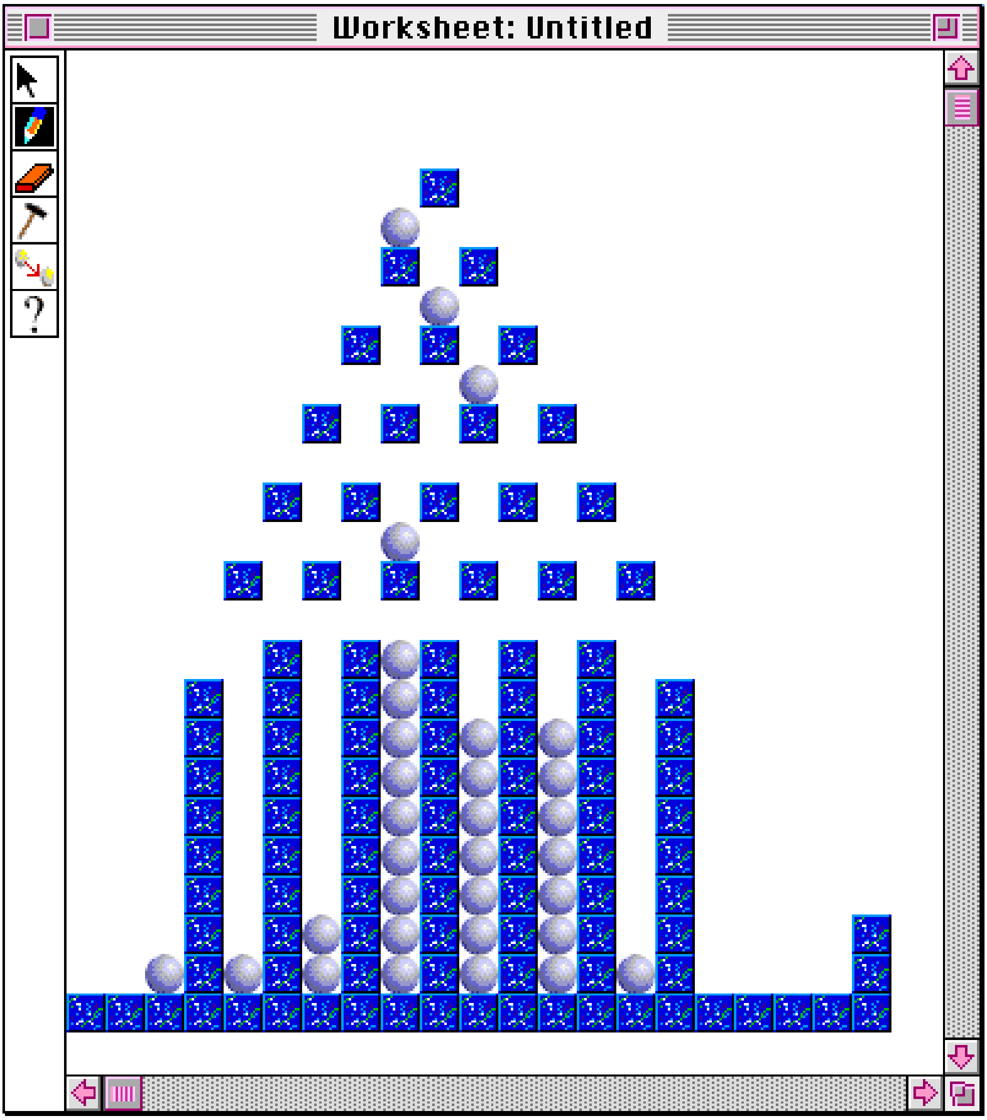


Figure 31: Particle World

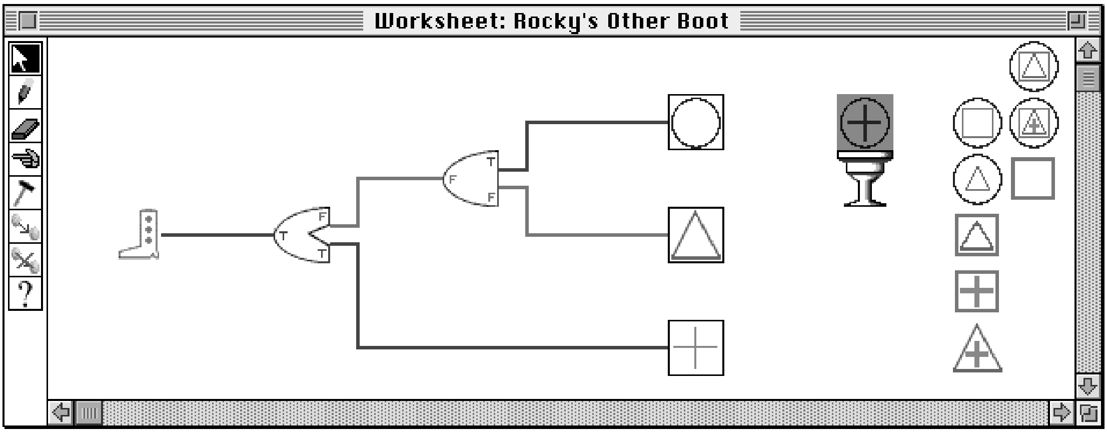


Figure 32: A Circuit Designed with Rocky's Other Boot

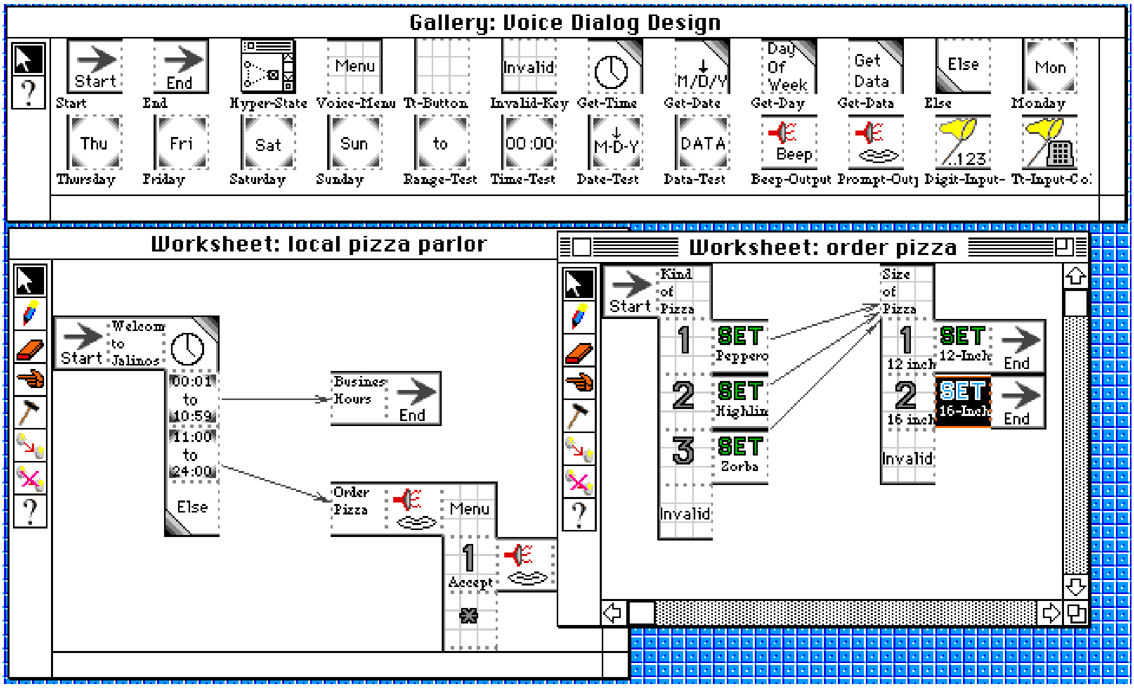


Figure 33: The Voice Dialog Design Environment

The design shown is an interface for a delivery service in a pizza parlor. If customers call outside of the restaurant's open business hours, they hear a standard message. If customers call during business hours, they can navigate through a series of voice menus to specify their pizza order. The design shown consists of two programs: a subprogram processing the incoming call based on business hours and a subprogram for specifying the desired pizza order.

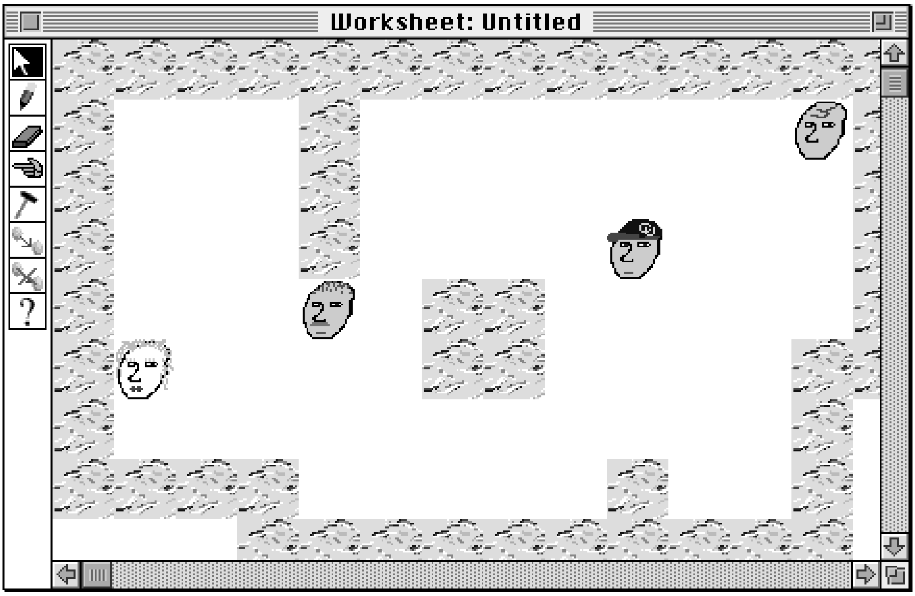


Figure 34: An Agent Party

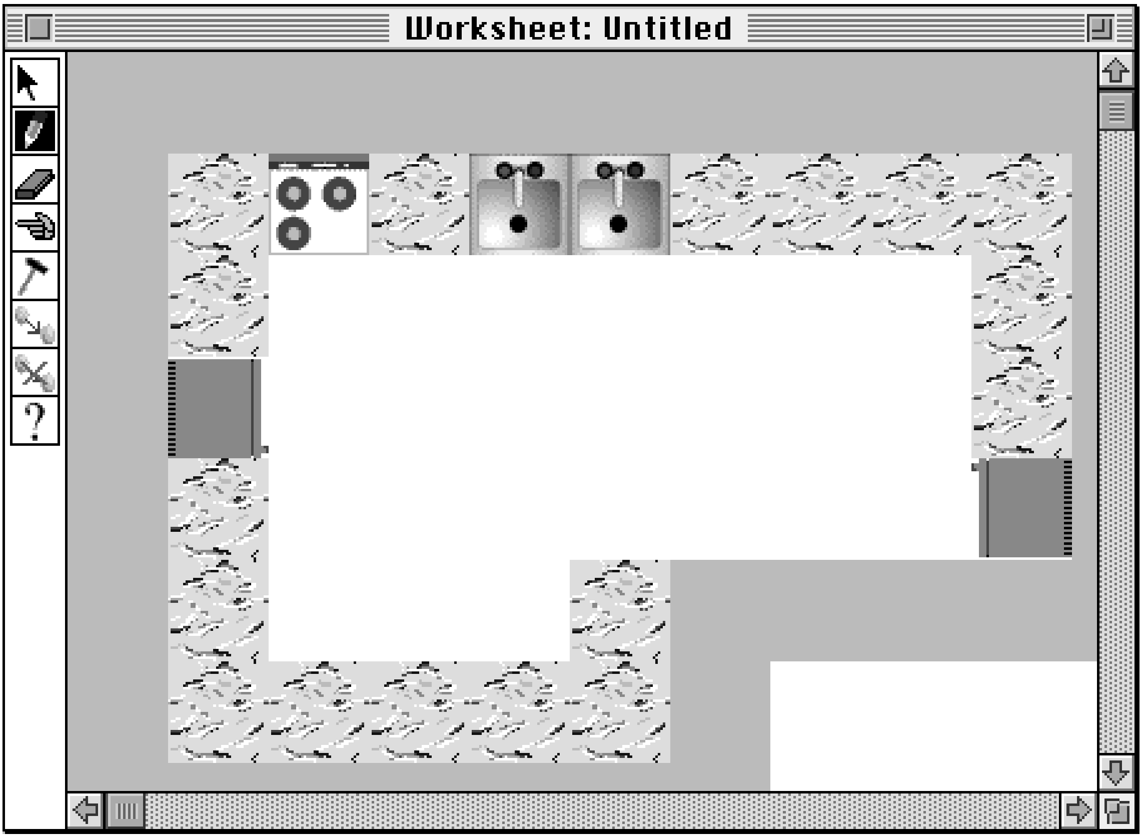


Figure 35: A Kitchen

game is to collect dollar bills in a maze. Initially, a player will use a manual agent and use the cursor keys to maneuver the agent through the maze to pick up the dollar bills. In a second phase of the game users select automatic agents with built-in behaviors. The most primitive agent uses a random walk approach that is not very efficient to pick up the dollar bills. Agents featuring more sophisticated approaches can be selected. Users can play with the different approaches, contrast them, and eventually program their own automatic agent with, hopefully, improved behavior (Figure 36). Learning opportunities:

- ***Learning through Programming.*** Search: what are good search algorithms?
- ***Learning about Programming.*** How can object-oriented programming be used to selectively refine search algorithms?
- ***Learning by Using.*** The use of momentum improves random search considerably.

5.11. EcoOcean

EcoOcean (Figure 37) deals with creatures living in the ocean, such as whales, sharks, and krill. Different types of water attract different types of animals. The system is used to experience the careful balance of nature as well as predator-prey relationships.

Learning opportunities:

- ***Learning through Programming.*** What is the life cycle of a creature; how do different life cycles influence each other?
- ***Learning about Programming.*** How can objects be parameterized so that end-users can introduce their own types of creatures?
- ***Learning by Using.*** Why do certain whales move into shallow water?

5.12. Village of Idiots

The Village of Idiots (Figure 38), inspired by Rich Gold, is a quite complex Microworld consisting of a village inhabited by idiots. The Village could be viewed as a zoomed-in version of SimCity. Individual idiots can move around in the city, meet other idiots, have sex and children, and they can die. Additionally, idiots who are limited in their movement by other idiots, e.g., in the case of overpopulation, can become aggressive to a point were they start to kill other idiots.

One design objective for building interesting villages is to have a relatively stable population. Villages either tend to get overpopulated, leading to a large number of killer idiots, or the idiots die prematurely.

One student extended the Village of Idiots to Maslow's Village populated by agents modeling Maslow's theory of the hierarchy of needs (Schultz, 1976). Initially, these agents satisfy just basic needs such as getting food. Over time, they can move up levels of needs to the point were they become self-actualized and start to help other agents.

Learning opportunities:

- ***Learning through Programming.*** Personality: how can human behavior be modeled as a hierarchy of needs (Schultz, 1976)?
- ***Learning about Programming.*** Behavior-based AI: how can a subsumption architecture be used to implement Maslow's model of personality?
- ***Learning by Using.*** Hungry people can be cruel.

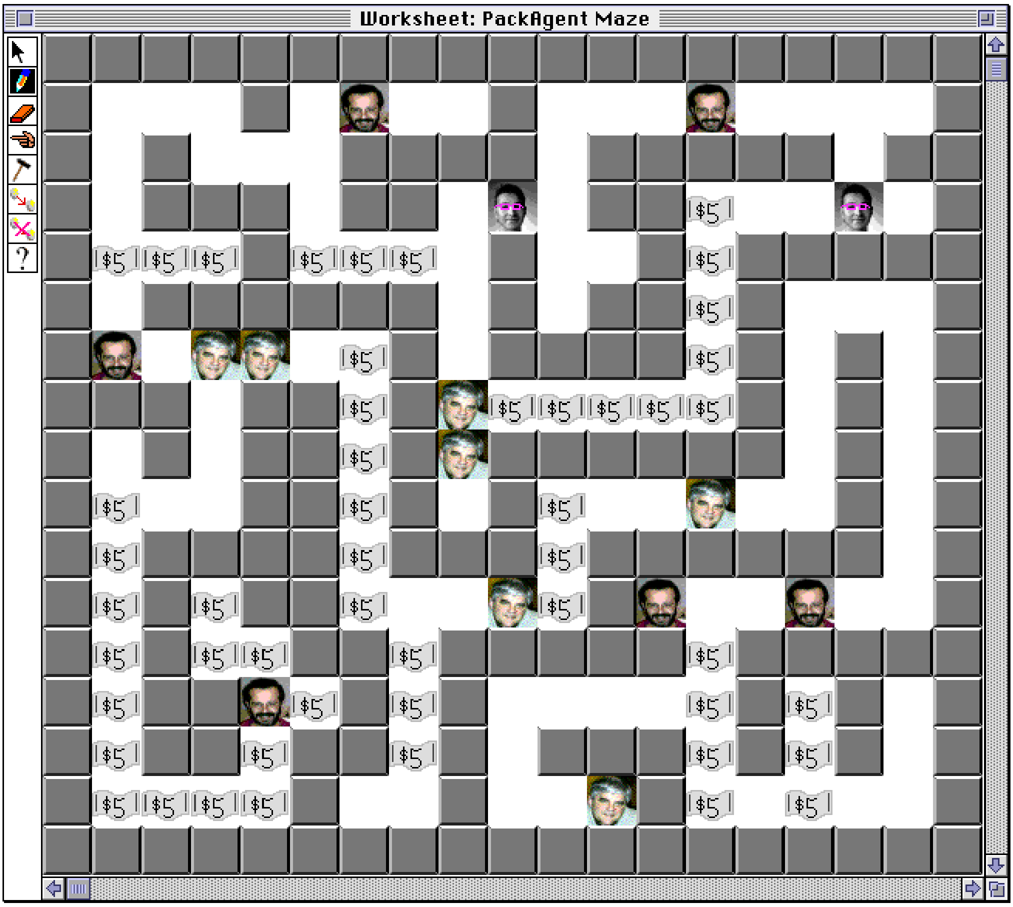


Figure 36: Manual and Automatic Money Collecting Agents in a Maze

6. CONCLUSIONS

The quest for the one ideal interactive learning environment (ILE) needs to be replaced with research concerning programming substrates that support the design and implementation of ILEs that are geared toward people solving specific problems. Agentsheets is a programming substrate to create ILEs. The visual formalism of Agentsheets, consisting of autonomous agents arranged in a grid, is a versatile framework that can be used across many different problem domains. By including incremental mechanism to define look and behavior, Agentsheets endorses collaborative design of ILEs and facilitates dialog between designers and end-users. Participatory theater allows end-users of ILEs created with Agentsheets to delegate repetitive, time-consuming, or tedious tasks to autonomous agents. The combination of visual formalism, collaborative design, and participatory theater supports the design and implementation of effective ILEs.

Programming substrates create learning opportunities: learning through programming, learning about programming, and learning by using a program. Our experience with Agentsheets indicates that is often difficult to predict and even more difficult to control the specific category in which learning will occur. This can represent a problem in educational approaches based on the “let the students do X to learn Y” strategy because we will not be able to determine what Y will be nor how we can evaluate that students really master Y. The approach taken with Agentsheets is to create opportunities to learn by enabling students to build and to use engaging problem-solving tools. We believe that if tools are engaging and if they provide learning opportunities then students will be motivated to learn.

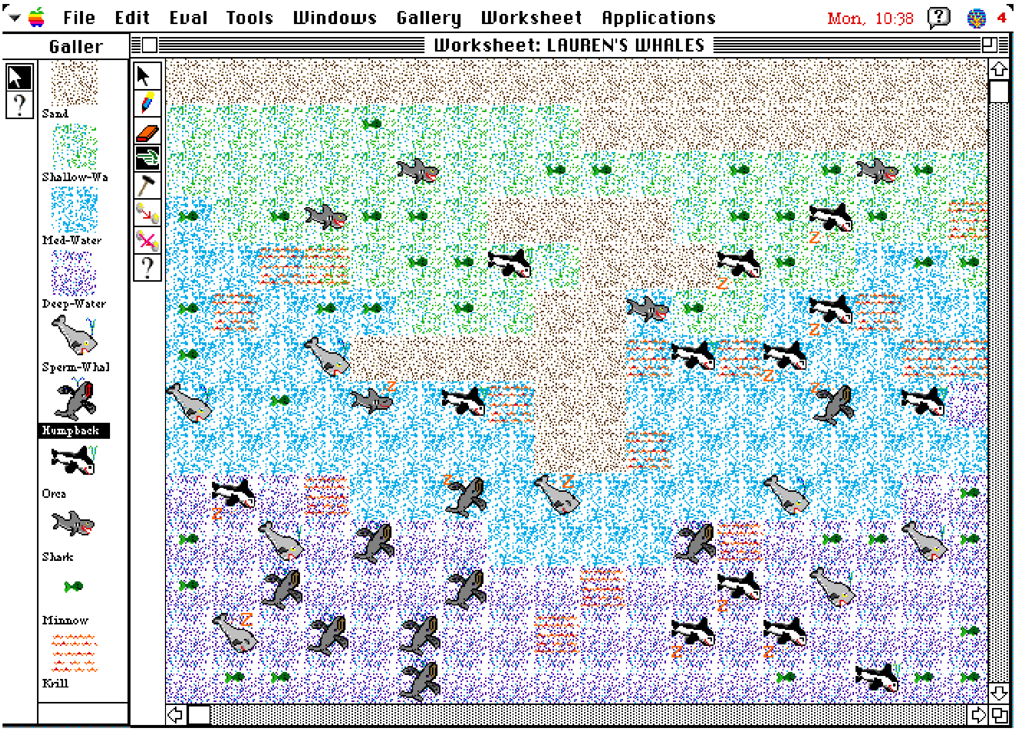


Figure 37: EcoOcean

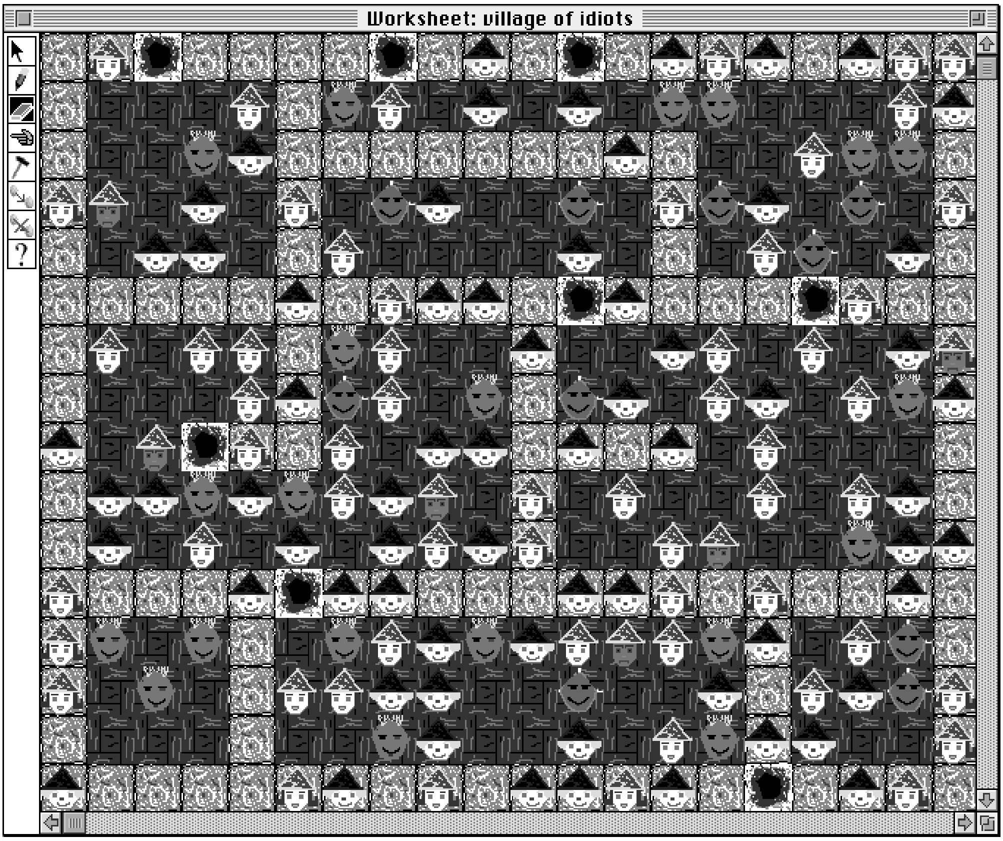


Figure 38: Village of Idiots

8. REFERENCES

Bell, B. (1991). ChemTrains: A Visual Programming Language for Building Simulations (Technical Report No. CU-CS-529-91). Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado.

Bell, B. (1992) Using Programming Walkthroughs to Design a Visual Language. Departement of Computer Science, University of Colorado at Boulder, Boulder, Colorado.

Bell, B., Rieman, J., & Lewis, C. (1991). Usability Testing of a Graphical Programming System: Things We Missed in a Programming Walkthrough. In S.P. Robertson, G. Olson, & J. Olson (Eds.), Proceedings CHI'91, (pp. 7-12). New Orleans, LA: ACM Press.

Brooks, F. P. Jr., (1987). No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer, pp. 10-19.

Chang, S.-K. (1990). Principles on Visual Programming Systems. Englewood Cliffs, NJ: Prentice Hall.

Cypher, A. (1993). Watch What I Do: Programming by Demonstration. Cambridges: MIT Press.

Dewdney, A. K. (1990). The Magic Machine. New York: W. H. Freeman and Company.

Fischer, G., & Lemke, A. C. (1988). Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication. HCI, 3, pp. 179-222.

Fischer, G., Lemke, A. C., Mastaglio, T., & Morch, A. (1991). The Role of Critiquing in Cooperative Problem Solving. ACM Transactions on Information Systems, 9(2), pp. 123-151.

Furnas, G. W. (1991). New Graphical Reasoning Models for Understanding Graphical Interfaces. In S.P. Robertson, G. Olson, & J. Olson (Eds.), Proceedings CHI'91, pp. 71-78. New Orleans, LA: ACM Press.

Gaver, W. W. (1991). Technology Affordances. In S. P. Robertson, G. Olson, & J. Olson (Eds.), Proceedings CHI'91, (pp. 79-84.) New Orleans, LA: ACM Press.

Golin, E. J. (1991). Tool Review: Prograph 2.0 from TGS Systems. Journal of Visual Languages and Computing, (2), pp. 189-194.

Green, T. R. G., Petre, M., & Bellamy, R. K. E. (1991). Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the 'Match-Mismatch' Conjecture. In Koeneman-Belliveau (Ed.), Empirical Studies of Programmers: Fourth Workshop, (pp. 121-146). Norwood, NJ: Ablex Publishing Corporation.

Guttag, J. V. (1991). Why Programming Is Too Hard and What to Do About It. In A. Meyer, J. Guttag, R. L. Rivest, & P. Szolovits (Eds.), Research Directions in Computer Science: An MIT Perspective (pp. 9-28). Cambridge, MA: MIT Press.

Kurlander, D., & Feiner, S. (1992). A History-Based Macro By Example System. In Proceedings of the ACM Symposium on User Interface Software and Technology, (pp. 99-106). Monterey, CA: ACM Press.

Lai, K.-Y., Malone, W. M., & Yu, K.-C. (1989). Object Lens: A "Spreadsheet" for Cooperative Work. ACM, 6(4), 332-353.

Laurel, B. (1993). Computers as Theater. Reading, MA: Addison-Wesley Publishing Company.

Lewis, C., & Olson, G. M. (1987). Can Principles of Cognition Lower the Barriers to Programming? In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop, (pp. 248-263). Norwood, NJ: Ablex Publishing.

Lieberman, H. (1987). An Example-Based Environment for Beginning Programmers. In R. W. Lawler & M. Yazdani (Eds.), Artificial Intelligence and Education (pp. 135-151). Norwood, NJ: Ablex Publishing.

McIntyre, D. W., & Glinert, E. P. (1992). Visual Tools for Generating Iconic Programming Environments. In Proceedings of the 1992 IEEE

Workshop on Visual Languages, (pp. 162-168). Seattle: IEEE Computer Society Press.

Myers, B. A. (1988). The State of the Art in Visual Programming and Program Visualization (Tech Report No. CMU-CS-88-144). Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

Nardi, B. (1993). A Small Matter of Programming. Cambridge, MA: MIT Press.

Nardi, B., & Zarmer, C. (1993). Beyond Models and Metaphors: Visual Formalisms in User Interface Design. Journal of Visual Languages and Computing(4), 5-33.

Negroponte, N. (1991). Beyond the Desktop Metaphor. In A. Meyer, J. Guttag, R. L. Rivest, & P. Szolovits (Eds.), Research Directions in Computer Science: An MIT Perspective (pp. 183-190). Cambridge, MA: MIT Press.

Papert, S. (1980). Mindstorms: Children, Computers and Powerful Ideas. New York: Basic Books.

Repenning, A. (1991a). Creating User Interfaces with Agentsheets. In V. Kumar & E. A. Unger (Ed.), 1991 Symposium on Applied Computing, (pp. 190-196). Kansas City, MO: IEEE Computer Society Press, Los Alamitos.

Repenning, A. (1991b). The OPUS User Manual (Technical Report No. CU-CS-556-91). Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado.

Repenning, A. (1993) Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments. Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado.

Repenning, A. (1994). Bending Icons: Syntactic and Semantic Transformation of Icons. To appear in: Proceedings of the 1994 IEEE/CS Symposium on Visual Languages, . St. Louis, MO: IEEE Computer.

Repenning, A., & Citrin, W. (1993). Agentsheets: Applying Grid-Based Spatial Reasoning to Human-Computer Interaction. In 1993 IEEE Workshop on Visual Languages, (pp. 77-82). Bergen, Norway: IEEE Computer Society Press.

Repenning, A., & Sumner, T. (1992). Using Agentsheets to Create a Voice Dialog Design Environment. In H. Berghel & E. Deaton (Eds.), Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing, (pp. 1199-1207). Kansas City, MO: ACM Press.

Repenning, A., & Sumner, T. (1994). Programming as Problem Solving: A Participatory Theater Approach. To Appear in: Workshop on Advanced Visual Interfaces ‘94, Bari, Italy.

Resnik, M. (1992) Beyond the Centralized Mindset: Explorations in Massively-Parallel Microworld. Computer Science, Massachusetts Institute of Technology.

Schultz, D. (1976). Theories of Personalities. Monterey, CA: Brooks/Colr Publishing Company.

Shu, N. (1988). Visual Programming. New York: Van Nostrand Reinhold Company.

Smith, D. C., Cypher, A., & Spohrer, J. (1994). KidSim: Programming Agents Without a Programming Language. Communications of the ACM, 37(7), 54-68.

Toffoli, T., & Margolus, N. (1987). Cellular Automata Machines. Cambridge, MA: MIT Press.