# A Graphical Programming Interface for a Children's Constructionist Learning Environment

by

Andrew C. Cheng

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 22, 1998
[June 1998]

Author [illegible]
Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by [illegible]
Mitchel Resnick
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

# A Graphical Programming Interface for a Children's Constructionist Learning Environment

by

Andrew C. Cheng

Submitted to the Department of Electrical Engineering and Computer Science

May 22, 1998

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

## ABSTRACT

Pet Park Blocks is a graphical programming interface for children. It is designed to lower the cognitive threshold for children to begin programming in Pet Park, a graphical online virtual universe created by Austina De Bonte. Pet Park Blocks is a good interface for beginners in Pet Park to learn basic programming concepts. The interface also provides a smooth transition to the original textual Pet Park programming environment. In this sense, Pet Park Blocks effectively provides scaffolding to help users think at one level and transition smoothly to the next.

Thesis Supervisor: Michel Resnick
Title: Associate Professor, MIT Media Laboratory

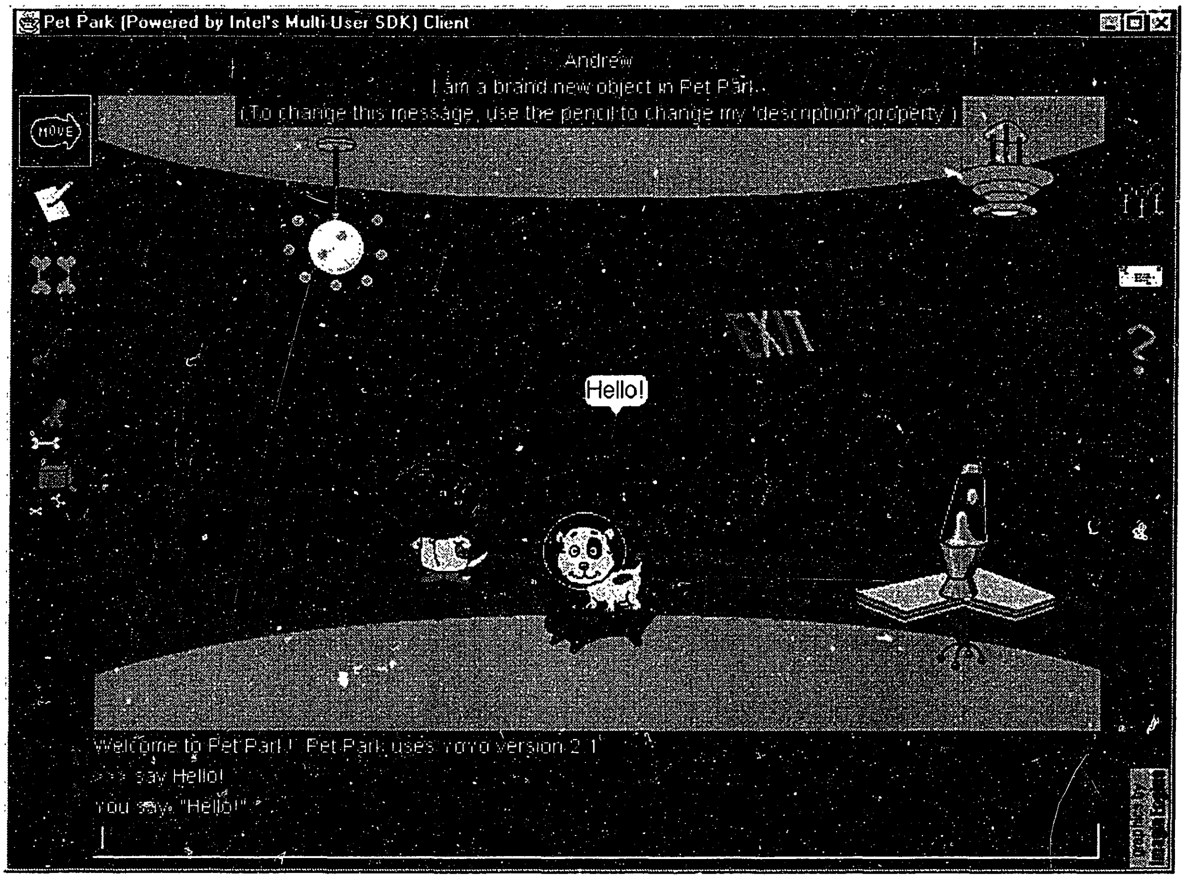
# Table of Contents

# 1. Introduction

In recent years, educators have experimented with a new theory of learning called constructionism. Under constructionism, children learn by choosing projects of their own that involve design, and by sharing their designs (Papert 1993). Pet Park is a graphical online virtual universe for children which emphasizes constructionist learning. It is a computer realm where children can log on, create their own animated pets, and program them to have animated sequences using a child-friendly version of Java called YoYo (Begel 1997). Pet Park Blocks is a graphical programming interface that accompanies Pet Park's original textual programming interface. This graphical interface is intended to help the younger users start programming and to help them make the transition to textual programming. This cognitive assistance is called scaffolding. This thesis describes the design goals and implementation of Pet Park Blocks. Pet Park Blocks can be best understood after a more thorough explanation of Pet Park and its focus on constructionism.

## 1.1 Description of Pet Park

Pet Park is a graphical online virtual universe for kids. Each child is represented by a graphical entity called an avatar. Figure 1 shows two avatars in a Pet Park location called "The Hotel." The one in the chair is named Andrew and he inherits from a basic Pet Park creature called Spotnik. All creatures that inherit from Spotnik look the same. This one is saying, "Hello!" to the other avatar. A child can move his avatar from one virtual location to another by selecting exits and other shortcuts. For example, in Figure 1 you can see the Exit that looks like a doghouse. This is an exit to the Library. Children can communicate to each other by typing messages that are seen by every other user within the same location. The messages show up as text balloons above the avatar that said them.



Andrew
I am a brand new object in Pet Park.
(To change this message, use the pencil to change my 'description' property.)

```
Welcome to Pet Park!  Pet Park uses yoyo version 2.1
>>> say Hello!
You say, "Hello!"
```

**Figure 1: A Pet Park location named "The Hotel"**

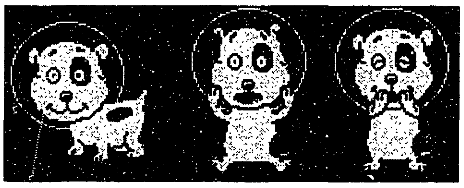


**Figure 2: Spotnik at ease, surprised, and laughing**

Children can create new objects to add to Pet Park, and program these objects with behaviors. A child can create a dog that jumps and says, "Let's have a party!" whenever three or more users are in the dog's location. Children can also program new animations for their avatars. For example, Spotnik, comes with 20 basic animation clips, or scripts. See Figure 2 for images of Spotnik. One script is called `Surprise`, and when it is called, it causes the Spotnik avatar to jump up, open its eyes wide and cover its mouth in surprise. Another script is called `Laugh`, and it causes Spotnik to laugh in the same standing position. A child can create a new animation for his Spotnik avatar, called `Congratulate`:

```
Surprise
Laugh
say "Wow!  That was great!  Congratulations!"
```

This script plays the `Surprise` and `Laugh` animations and ends with the avatar saying, "Wow! That was great! Congratulations!" All programming for Pet Park is done in a child-friendly programming language called YoYo (Begel 1997). YoYo incorporates many ideas from Java and from Logo.

## 1.2 Properties of Constructionism

Pet Park is a constructionist learning environment. Constructionism is a theory of learning that emphasizes design, ownership, and community.

### 1.2.1 Design

Each new animation is a design project. The process of design is rich with learning opportunities (Papert 1993). A child begins with a new animation concept and learns many things as he figures out how to organize his thoughts, how to express them to other children, how to realize them in the act of "building" a program, and how to fix the new program iteratively until he gets what he had in mind.

In Pet Park, design is done through programming. Seymor Papert argues that programming is an excellent constructionist learning activity (Papert 1980). When a child writes a program, the child has to express his ideas explicitly and in the organized fashion required by the programming language. This mode of expression is very conducive to experimentation; a child can execute his program to see how thoroughly and accurately he expressed his thoughts. In this way, programming is a way to make thought processes more explicit and concrete and therefore more accessible to reflection (Papert 1980).

### 1.2.2 Ownership

Each animation is a project of the child's own choosing. Each child feels a sense of ownership for his avatar because that avatar represents him or her to the other children. Therefore the child is highly motivated to improve continually the avatar with increasingly complex animations. The result is that the child is willing to spend time to overcome obstacles and figure out problems that stand in the way of the completion of a new animation script. The personal connection between the user and the avatar engenders ownership of the ideas that are learned in the design process.

### 1.2.3 Community

Each new animation is constructed in an environment where other children can give suggestions, assistance, and feedback. As a Pet Park user helps others, the user draws on the knowledge acquired in other projects. This process strengthens and fleshes out the user's grip on the knowledge, while creating new opportunities to learn about effective communication. Meanwhile, those who are receiving assistance are encouraged and further motivated in their own projects. The set of design projects within the community also provides fertile soil where new ideas can grow. Amy Bruckman has explored the way that a community supports constructionist activities, and the way that design projects enrich the community (Bruckman 1997).

## 1.3 Thesis Overview

Since much of the learning in Pet Park takes place while a child programs, special attention must be paid to the programming interface. Pet Park Blocks is intended to lower the cognitive threshold for children who want to join the community. The Pet Park Blocks programming interface makes it easy for younger users to write basic programs. It helps beginners learn some of the key concepts in programming and also provides a smoother transition to programming with a textual language. This idea of helping students at one level so that they can transition smoothly to the next level is called scaffolding. Pet Park emphasizes learning in a constructionist environment, and Pet Park Blocks helps younger and less experienced users get involved by providing scaffolding for the programming environment.

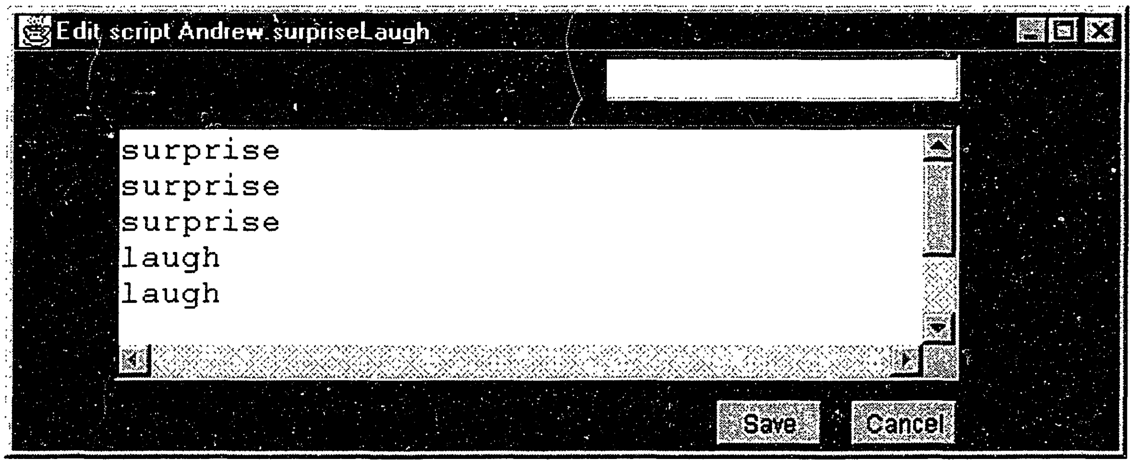
The next section of this thesis describes the challenges that a young programmer faces when using a textual programming interface. It also describes how a graphical programming interface can reduce those challenges and also how such an interface can provide useful aides to thinking. The third section of this thesis is a brief discussion of several existing graphical programming languages and how they compare to the needs of young Pet Park users. The fourth section describes the features of Pet Park Blocks that are intended to provide scaffolding to users. Many of the features are borrowed from the other graphical programming languages. Finally, the fifth section lays out the current implementation of Pet Park Blocks.

# 2. Textual and Graphical Programming Languages

Both textual and graphical programming languages have their advantages and disadvantages. However, this thesis is only concerned with the programming environment as experienced from the point of view of young and inexperienced programmers.

## 2.1 Challenges Inherent in Textual Languages and Addressed by Graphical Languages

Figure 3 shows the original Pet Park textual programming interface. All programming is done in a child-friendly version of Java called YoYo. YoYo presents the expressive power of Java in a simpler syntax that is strongly based on Logo. However, any textual programming language presents a young user with certain challenges.



```
surprise
surprise
surprise
laugh
laugh
```

**Figure 3: The original textual interface for Pet Park**

### 2.1.1 Complexity of Expression

The ideal programming environment would allow the users to focus on what he or she wants to express instead of how to express it. However, with a textual programming interface, a child spends much of his time figuring out how to say things in the language. For example, consider Figure 4. It shows the ways to express the same idea in three different languages. The idea is to print out a message: "`Hello World!`" Figure 4a shows how this is done in Java. Figure 4b is for YoYo, and Figure 4c is one graphical alternative.

```
(4a) System.out.println("Hello World!");

(4b) say "Hello World!"

(4c) Programmer is to type
     the desired text
     in the white balloon.
     This text will be
     printed to the screen
     verbatim.
```

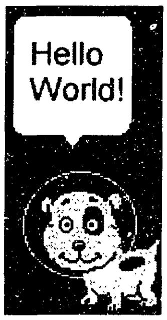


**Figure 4: Three ways to print "Hello World!"**

As one moves from 4a to 4c, one can see that there is less to remember in order to express the idea of saying, "Hello World!" In 4a, a child must remember to type the string `System.out.println` even though this string includes ideas that the child does not need to understand. YoYo abstracts away this excess. The concept in 4c moves even closer to a plain representation of the idea since the avatar stands with an empty text balloon waiting to be filled.

### 2.1.2 Syntax

The challenge of getting all the syntax correct in textual programming environments can be frustrating, especially for young users. This challenge is deeply linked with the previous one described above. Reconsider Figure 4. The YoYo code in 4b requires that the user type `say` before the message. One can imagine forgetting which of the following lines has the correct syntax:

```
say    Hello World!
say,   Hello World!
say,  "Hello World!"
say:   Hello World!
say:  "Hello World!"
```

Pet Park actually uses two variations. When programming scripts in YoYo, a text message must be enclosed in quotes. However, quotes are not necessary when a child types at the command prompt what he or she wants the avatar to say. This might confuse a young programmer who could have the preconception that there is one correct syntax. As he tries to figure out what the syntax is, the system allows for variations thus potentially causing momentary confusion: "Wait. I thought it was supposed to be done this way." The limited variation can cause further confusion: "I thought I could say it any way I wanted to. I thought the computer would figure out what to do." Eventually a child might wonder: "What would happen if I tried to include a quote mark in my message?" During a conversation with a teacher, the child might have some trouble understanding what an escape character is. Finally, the child could ask: "How would I include an escape character in my message?"

We can see here that the child is focusing on **how** to express an idea instead of **what** idea to express. While this focus can be fruitful in learning the differences between different programming languages, it is probably too esoteric for beginners in Pet Park.

Reconsider Figure 4c. The image suggests that whatever is typed in the text balloon will appear verbatim when it is executed, because the image is actually identical to the execution: some text will appear in a white balloon directly above the avatar. This approach virtually eliminates the need for escape characters and other syntax.

A program that is easy to compose is also easy to debug. Reconsider Figure 4. A Java syntax error is harder to find than a YoYo syntax error simply because Java programs have more syntax for a programmer to wade through. However in a graphical programming environment, errors can be prevented in real-time as the system notifies the user that a certain thing cannot be done: "Sorry, only blue program blocks can go here." This reduces the errors in programming to mostly conceptual errors: "Ah, I should put more `Laugh` blocks here because a single `Laugh` block doesn't represent as long of a laugh as I wanted." A child will be freed from syntax concerns such as: "Was `Laugh` supposed to be capitalized?" and will be able to think primarily of what concept is missing or out of sequential order.

### 2.1.3 Typing

One last challenge is typing. The first program I ever tried to type into a computer was one I found in the BASIC handbook for my Commodore64. The book said that the program listed on the page would cause the computer to display a flower pattern and play a nice melody. I took a long time in entering the program into the computer because I did not yet know how to type. Finally I executed the program and the computer displayed something like torn wallpaper and from the keyboard speaker came the sounds of two gunshots and a scream, played repeatedly. I spent another hour trying to correct any typing mistakes and the final result was a program that displayed a black screen and played no sound. I had no idea what many of the program lines meant, and, although that was a very deep and serious problem, my most pressing concern was rather that the endeavor was taking forever because I didn't know how to type. I quickly relapsed into my previous mode of playing Montezuma's Revenge, a video game. A graphical programming environment can drastically reduce the amount of typing.

## 2.2 Other Useful Virtues of Graphical Programming Languages

In addition to eliminating or reducing certain challenges of textual programming languages, graphical programming languages can provide useful visual cues.

### 2.2.1 Visual Metaphors

Graphical programming languages can represent certain common programming ideas in a straightforward manner that makes the meaning visually clear. The term "visual metaphor" is related to the ordinary meaning of "metaphor" which is: a figure of speech literally meaning one thing, but used in place of another to emphasize the similarity between them. A visual metaphor is a graphical convention for displaying a certain program construct, where the convention makes the meaning of the construct obvious by its similarity. For example, a branch can represent a conditional statement such as an if in a sequence of blocks, with the boolean test at the branch.

### 2.2.1.1 Metaphor for Parallelism

Another good example of a program construct that can be represented well by a visual metaphor is parallelism. Two segments of code that can execute in parallel can be depicted in a graphical programming environment as two geometrically parallel paths through which the control of the program flows. In Figure 5, the arrows indicate that the control runs along two paths simultaneously and rejoins at the end of the two parallel code segments.

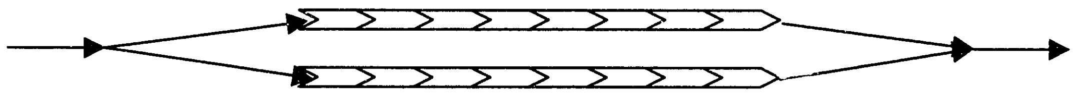


**Figure 5: A visual metaphor for parallelism**

### 2.2.2 Visual Cues

Another way a graphical programming interface can help the programmer is by providing visual cues. Two kinds of visual cues are described here.

### 2.2.2.1 Corresponding Shapes and Colors

In Pet Park, an animation such as `Congratulate` (see Section 1.1) is procedurally abstracted and represented by that name: `Congratulate`. Animations can be called one after another because every animation sequence begins and ends with the same image of the avatar. This convention allows one animation to leave off where another will begin, much the way cursive fonts on a computer have letters that begin and end at corresponding points. In Pet Park Blocks, blocks such as the one shown in Figure 6 represent animations.

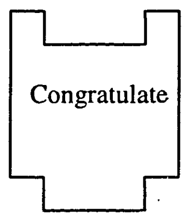


**Figure 6: A Congratulate animation block**

Notice the notch at the top and the protrusion at the bottom. Animation blocks can be connected in a vertical column with the protrusion of a block fitting into the notch of the next block as shown in Figure 7. The blocks fit together like jigsaw puzzle pieces. These notches and protrusions are a visual hint to the programmer that pieces that do not have a notch at the top cannot be connected at the bottom of animation blocks. For example, a boolean operator such as an and piece should not be connected to any animation block, since animation blocks have no boolean value for the operator to use. Therefore boolean operators are represented by graphical entities which do not have top notches and bottom protrusions. Figure 8 shows how boolean operators and variables fit together horizontally, because boolean operators have concave sides that accept the round convex sides of boolean variables.

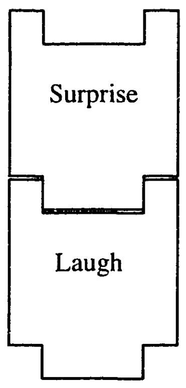


**Figure 7: How animation blocks fit together**

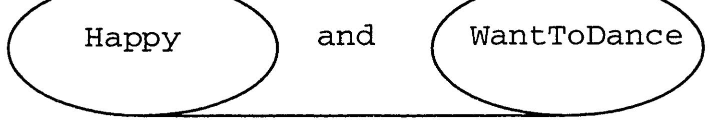


**Figure 8: How boolean variables and operators fit together**

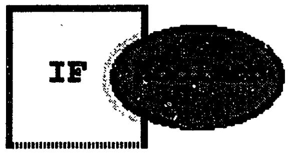


**Figure 9: Contact edge tinted**

Pieces that correspond to each other can be color-coded. For example, all boolean operators can be dark blue and boolean variables can be lighter blue. Also, the edge of an `if` block that is meant to hold a boolean expression can be tinted blue, indicating that only blue boolean blocks could go there. In Figure 9, the concave contact surface that the `if` statement has for boolean variables is tinted blue.

### 2.2.2.2 Message Icons for Visual Feedback

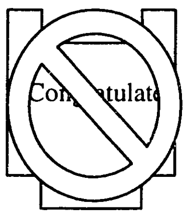


**Figure 10: Visual cue**

Another visual cue that graphical programming environments can provide is an occasional pop-up icon that notifies the user of what action will take place if the user proceeds with the current mouse operation. For example, if the user is dragging an animation block near a boolean operator, the system can determine that if the animation block were dropped there, it would not connect with the boolean operator. The system could then display a visual cue over the animation block, showing that it will be rejected if dropped there. See Figure 9 for an example of this visual cue.

### 2.2.3 Advanced Visual Cues

An animation block can have, on its face, a little window showing the actual animation sequence playing in a loop. This way, the block can be identified either by its text label or its contents, the animation. Many other visual cues and metaphors are possible.

## 2.3 Limitations of Graphical Programming are Acceptable

Although there are limitations to graphical programming, most limitations are acceptable in Pet Park Blocks because the interface is focused on providing scaffolding for beginner programmers. These programmers will write simple programs and for that reason either (1) they will not encounter these limitations or (2) the limitations will not have too negative an impact.

For example, one limitation is that graphical programming constructs usually take up more screen space than their text equivalents. Therefore, screen space places a limit on how much of the program the user can view at any given time. This limitation is acceptable for beginner programmers because Pet Park Blocks is designed so that programs tend to grow only in one direction: down. If a program does not fit on the screen in its entirety, the programmer only has to scroll the program in one dimension.

A second limitation is that graphical programming languages generally have less expressive power than textual programming languages. This disadvantage is acceptable for beginning users, however, since many basic ideas that they want express can be represented graphically. For example, a new animation that is simply the result of chaining other animations together can be easily expressed in Pet Park Blocks by a column of animation blocks.

As programmers grow in experience, their programs grow longer and require more expressive power. They discover that there are things that cannot be done in Pet Park Blocks. Therefore, they make the transition to the textual programming interface. Pet Park Blocks has certain features that are designed to make the transition a straightforward one. These features are discussed in Section 4.

## 3. Existing Graphical Programming Paradigms

The consideration of (1) several existing graphical programming paradigms and (2) how they compare to the needs of Pet Park programmers can provide a context for understanding the decisions made concerning the features, design, and implementation of Pet Park Blocks. The following graphical programming languages are categorized by the conceptual model they follow.

### 3.1 Data Flow and Control Flow

In the data flow paradigm, there are two basic graphical programming constructs: (1) lines or arrows showing the path that data takes during the execution of the program, and (2) blocks that represent transformations or operations that are done on the data. Consider Figure 10 for an illustration of this. The program's input is labeled as `Source`. This statement is operated on by transform `H1`. The result of this is operated on separately by transforms `H2` and `H3`, and the final `Answer` is the sum of the results from `H2` and `H3`. Notice that the program implies that the transformations done by `H2` and `H3` can be done simultaneously. This is another example of the use of the visual metaphor for parallelism depicted in Figure 5.

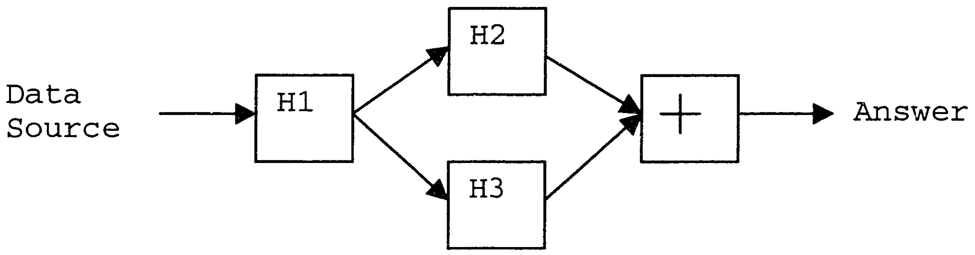


**Figure 11: Data flow example**

This programming style is useful for scientific applications where operations and transformations are done on data. However, for the purposes of Pet Park, the data flow paradigm is inappropriate. The basic graphical programming constructs are not transformations or operations but rather blocks that represent animation sequences. The reason behind this is that YoYo is a control flow language. Programs are sequences of actions that are occasionally modified by control structures such as loops and conditional statements. Data is not passed from one operator to the next, but rather the control of the environment flows through successive statements. Since Pet Park Blocks provides scaffolding for beginner programmers to gradually transition to the YoYo textual programming environment, Pet Park Blocks adheres to the control flow paradigm.

## 3.2 Rule-Based Environments: Agentsheets and Visual AgenTalk

Agentsheets (Repenning 1993) is a graphical language that compiles to Java. It was developed by Alex Repenning at the University of Colorado. In Agentsheets, an agent is an entity that is programmed with the ability to perceive and react to changes in its environment. Agents are organized on a grid called an "agentsheet." They communicate with each other directly and also according to their relative position on the agentsheet. For example, a programmer can create an agentsheet that represents the ecosystem of a pond. The agents on this agentsheet represent the individual fish. Each agent (or fish) can react to its environment by avoiding collisions with other fish, swimming towards food, etc.

Visual AgenTalk is a rule-based graphical programming environment for Agentsheets. Each agent can be programmed with a set of rules such as: "If the space on the agentsheet grid immediately to my right is empty, then move to that space." This is represented by the graphical rule depicted in Figure 12.

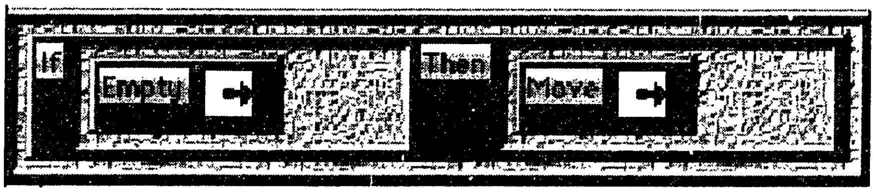


**Figure 12: An Agentsheets rule**

In the example of the pond ecosystem, a fish agent can have any number of rules. The agentsheet is associated with a virtual clock and with each passing clock tick, the system goes through each agent and determines which rules to fire. Suppose a given fish has five rules that are satisfied simultaneously on a given clock tick. The system chooses which rules to fire by consulting the heuristic supplied by the programmer. One heuristic is to choose one rule at random and fire it. Another heuristic is to impose an arbitrary order and fire the first three. The agentsheet goes through this procedure for each agent.

This may sound similar to Pet Park, because Pet Park also has agents that can react to the environment. However, there is a fundamental difference in the way the behavior of an agent is specified in the two environments. In Agentsheets, an agent obeys a set of specified rules. In Pet Park, agents follow an animation sequence specified in YoYo code whenever the user and requests that the animation be activated. The entire animation is a sequential matter and is not driven on each clock-tick by a set of rules.

## 3.3 Programming by Example: Cocoa

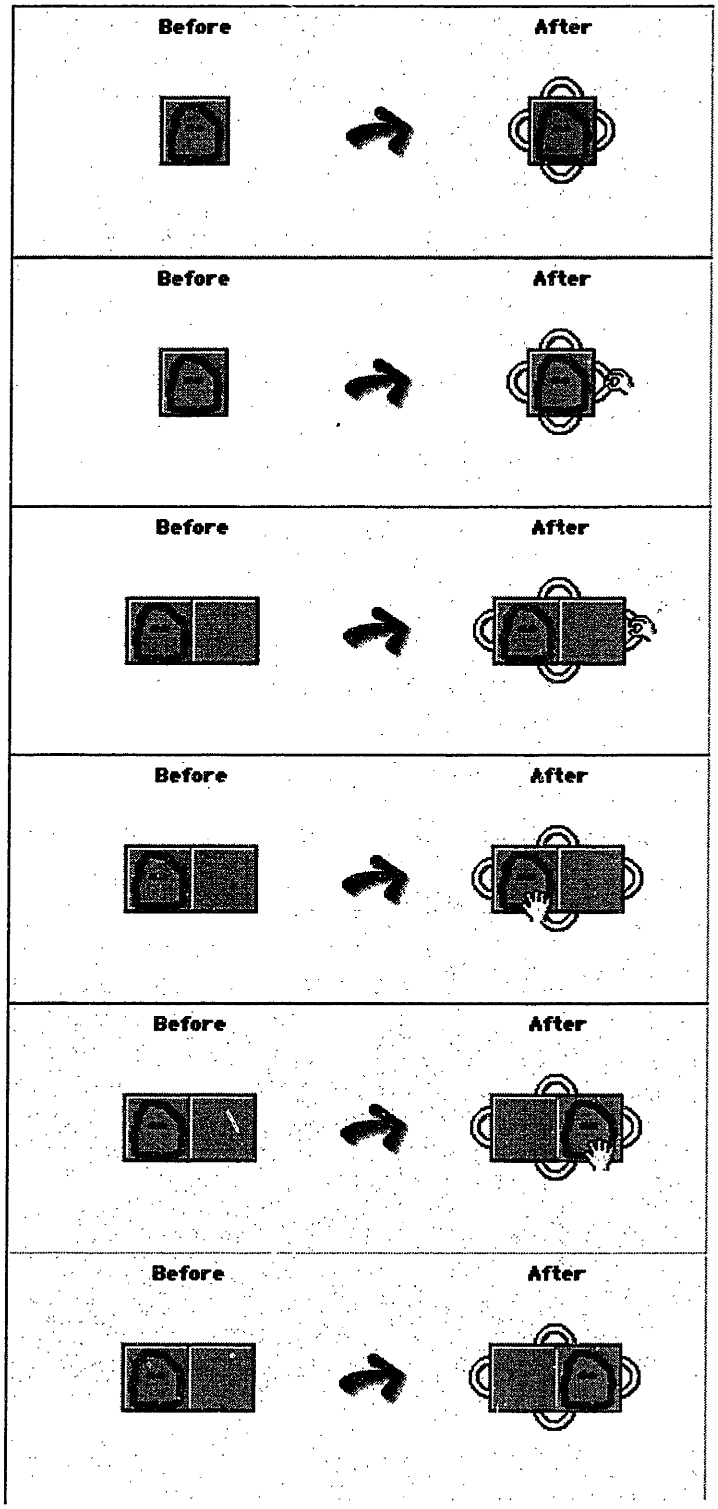


Figure 13: Creation of a Cocoa rule

Cocoa is a language developed at Apple. It is actually based on an early version of Agentsheets. Cocoa is also rule-based, but it is still noteworthy in this discussion because it emphasizes another programming paradigm: programming by example. In this paradigm, the programmer shows the computer what to do by giving it an example. The computer then figures out what the equivalent program code should be. Programming by example is also called programming by demonstration. Figure 13 shows the creation of a Cocoa rule that is the equivalent of the Agentsheets rule shown in Figure 12. The bottom image is the final form of the rule. If the space immediately to the right of the agent is empty, then the agent should move to that empty space. In order to create this rule, the user clicks on a `New Rule` button. The top image in Figure 13 is presented to the programmer, and each successive image shows the steps in creating the rule. The agent is in a single grid square in a `Before` stage and an `After` stage. The programmer needs to demonstrate to the computer that, if there is an open space to the right of the agent, the agent should move to that space. This demonstration is done by altering the `After` stage: the programmer needs to create an empty space to the right and then show how the agent should move there, all by interacting only with the `After` stage. The `After` stage has round handles or tabs on the edges that can be dragged by the mouse. The programmer drags the right handle one space to the right. The agent is now next to an empty grid space to its right. The programmer then drags the agent from its original location in `After`, to the empty space immediately to the right. Notice the behavior of the `Before` stage: it mimics all the changes in the `After` stage except any changes that involve the agent itself (such as a change in the agent's location).

Programming by example is a very intuitive way of specifying rules. Pet Park animations could be programmed this way if the environment supported the recording of macros. However, the original textual programming environment involved writing YoYo programs that call one animation after another.

## 3.4 Control Flow: LogoBlocks

LogoBlocks (Begel 1996) was developed at the Media Laboratory at MIT for the Cricket, a tiny computer that can communicate through two sensors and can control two motors. LogoBlocks emphasizes control flow. The graphical user interface consists of a workspace and a set of palettes. Each palette holds program pieces in the form of shapes with notches and protrusions that fit with other compatible pieces. The user builds a program by dragging blocks from the palette to connect them together in the workspace.

Figure 14 shows a short LogoBlocks program. Notice that the `or` block has two curved concave sides that correspond to the convex sides of the boolean blocks `SwitchA` and `SwitchB`. The blocks are also color-coded: control structures such as `if-thens` are yellow, actions such as `Beep` are green, and so on. The color scheme and the block shape convention are visual cues that show which blocks can fit together. They are useful visual cues that Pet Park Blocks borrows.

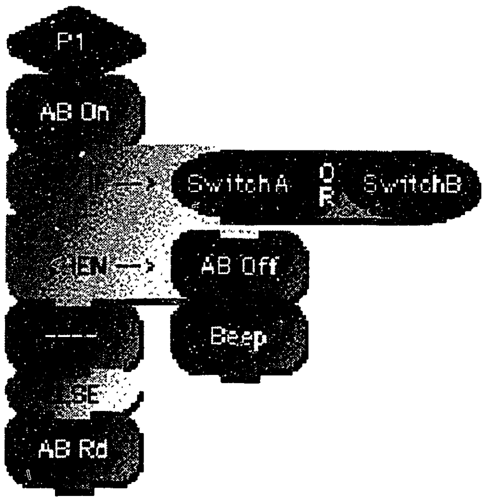


**Figure 14: A LogoBlocks Program**

# 4. Features of Pet Park Blocks

The following features of Pet Park are designed to help beginner programmers get their feet wet in programming in Pet Park so that they can later make a smooth transition to programming in the textual programming interface with YoYo. This section describes both the intended features as well as the ones that are actually implemented. See Figures 15 and 16 for an image of the current implementation of Pet Park Blocks. Figure 15 is the graphic mode of the programming interface. The right side displays the current palette. Two palettes are available containing the animation blocks belonging to `this` Pet Park creature and its `parent`. The left side is the workspace where the program is constructed. A program is built by dragging blocks from the palette to the workspace where they are added to the growing program. In Figure 15, the programmer is working on a `dance` script. So far, the script includes two animation blocks called `hop` and `wave`. Figure 16 shows the new textual programming interface. It still displays the palettes on the right. A user can click on an animation block in a palette and then click in the text window on the left and the code for that animation bock is copied into the program at that point.

## 4.1 Basic Visual Style: Control Flow

Pet Park Blocks borrows the intuitive visual style used in LogoBlocks: a program is formed by placing blocks together on the screen. Each block represents some bit of YoYo code, such as a short animation. An animation is a sequence of images. When an animation block is executed, the images are displayed in order on the screen where the Pet Park avatar is. The first and last images of each animation sequence are the same as the default image for the avatar. This way, blocks can be connected in the sequence and

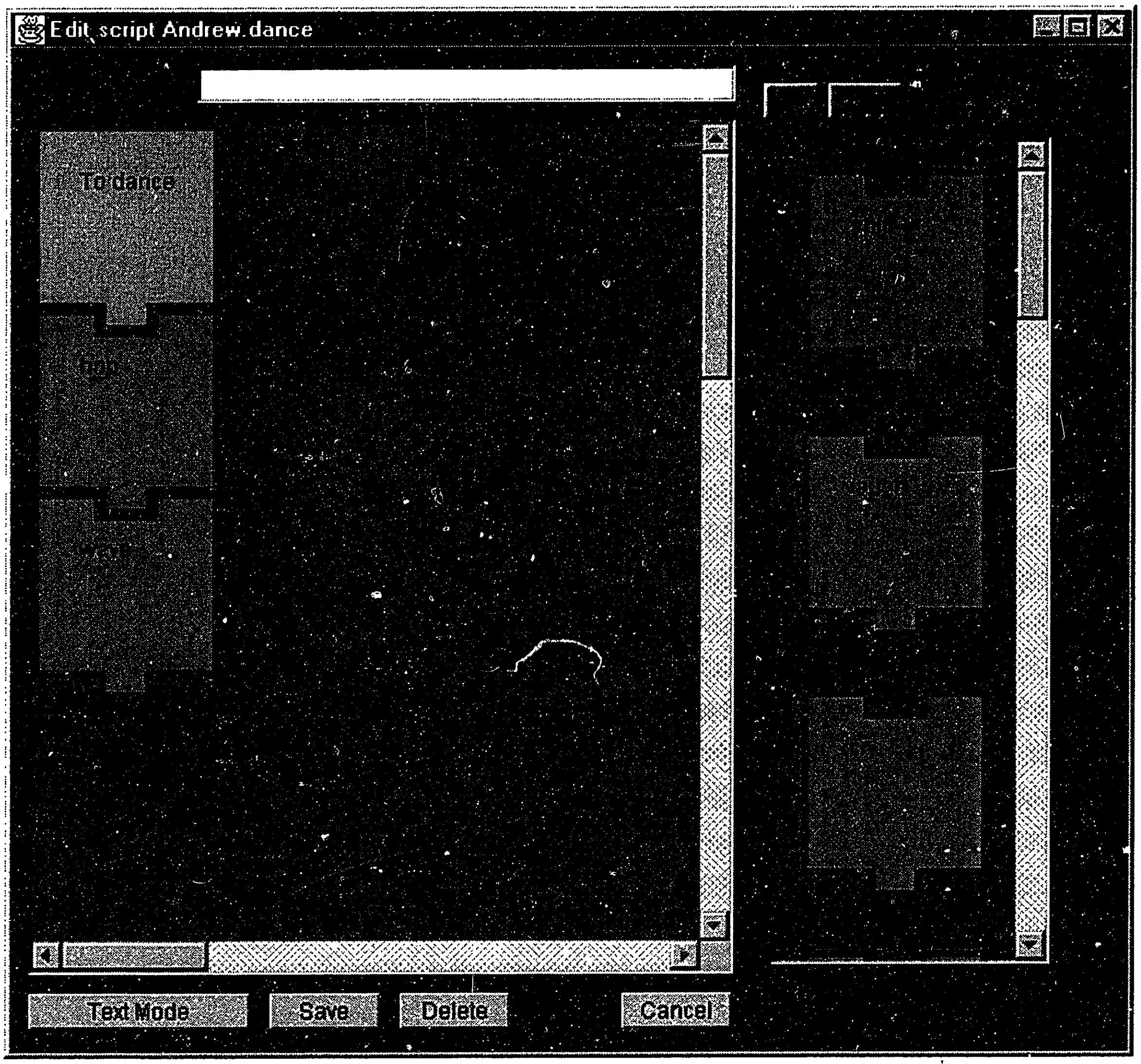
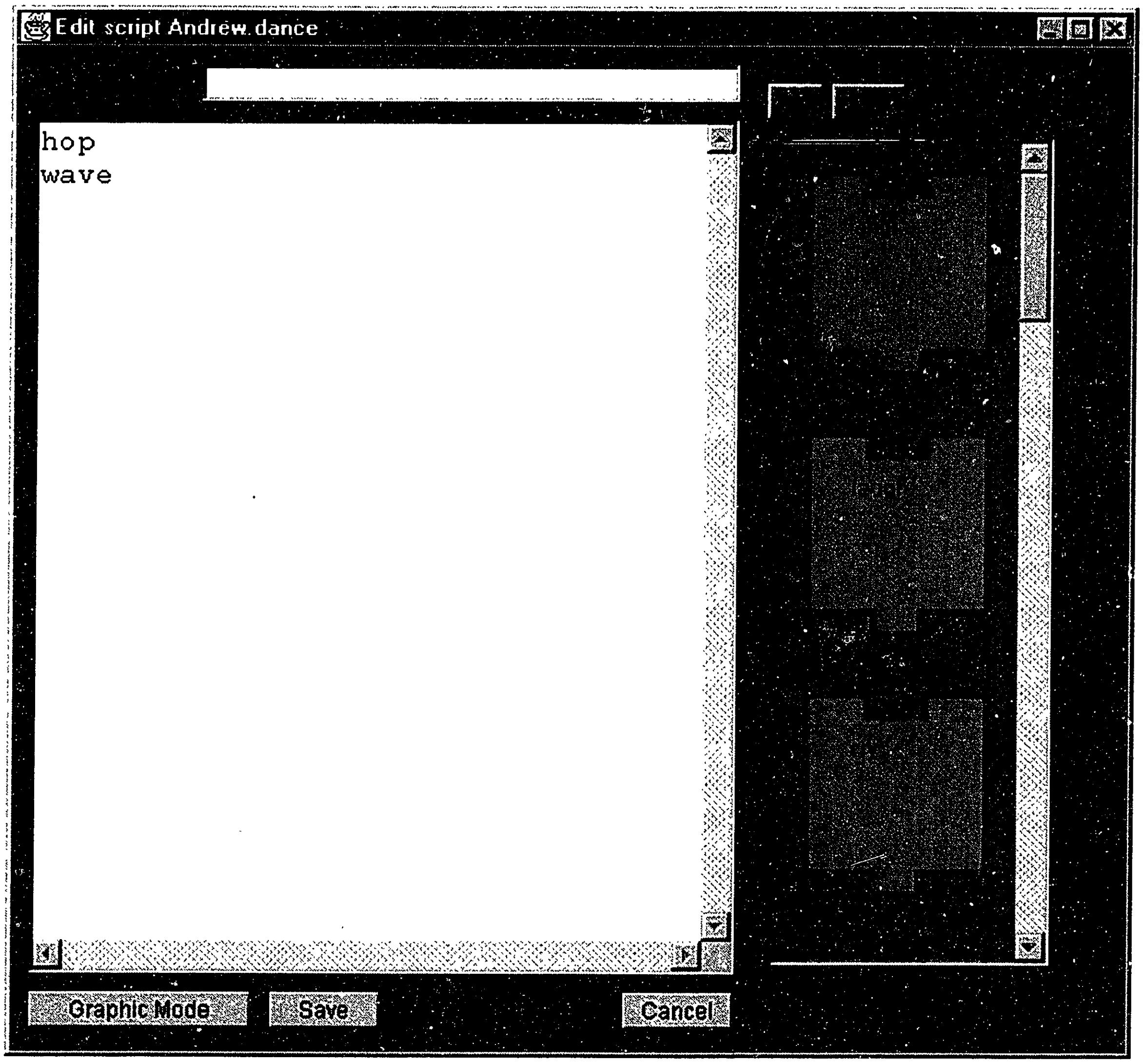


**Figure 15: The Pet Park Blocks graphical interface**

executed; each block leaves off where the next block begins. This convention is borrowed from cursive fonts: each letter ends at the same point so that the next letter can connect to it smoothly.

Each agent has his own library of animation blocks. For example, a child may be represented by an avatar named Stacy. Stacy's library of animations is initially empty. Whenever the child creates a new animation for Stacy, this animation is added to Stacy's library. When the child is editing or creating an animation, these libraries as well as that



```
hop
wave
```

**Figure 16: The new text interface**

of the agent's parent-creature appear in palettes. In the example, suppose that Stacy, the avatar, is a Pet Park creature that inherits from the basic Pet Park creature called Spotnik. The child can drag blocks from this palette to the workspace and connect it to the existing program.

Each animation block has a notch at the top and a protrusion of the bottom like the Logo Blocks `ABoff` and `Beep` shown in Figure 14. The blocks form a sequence

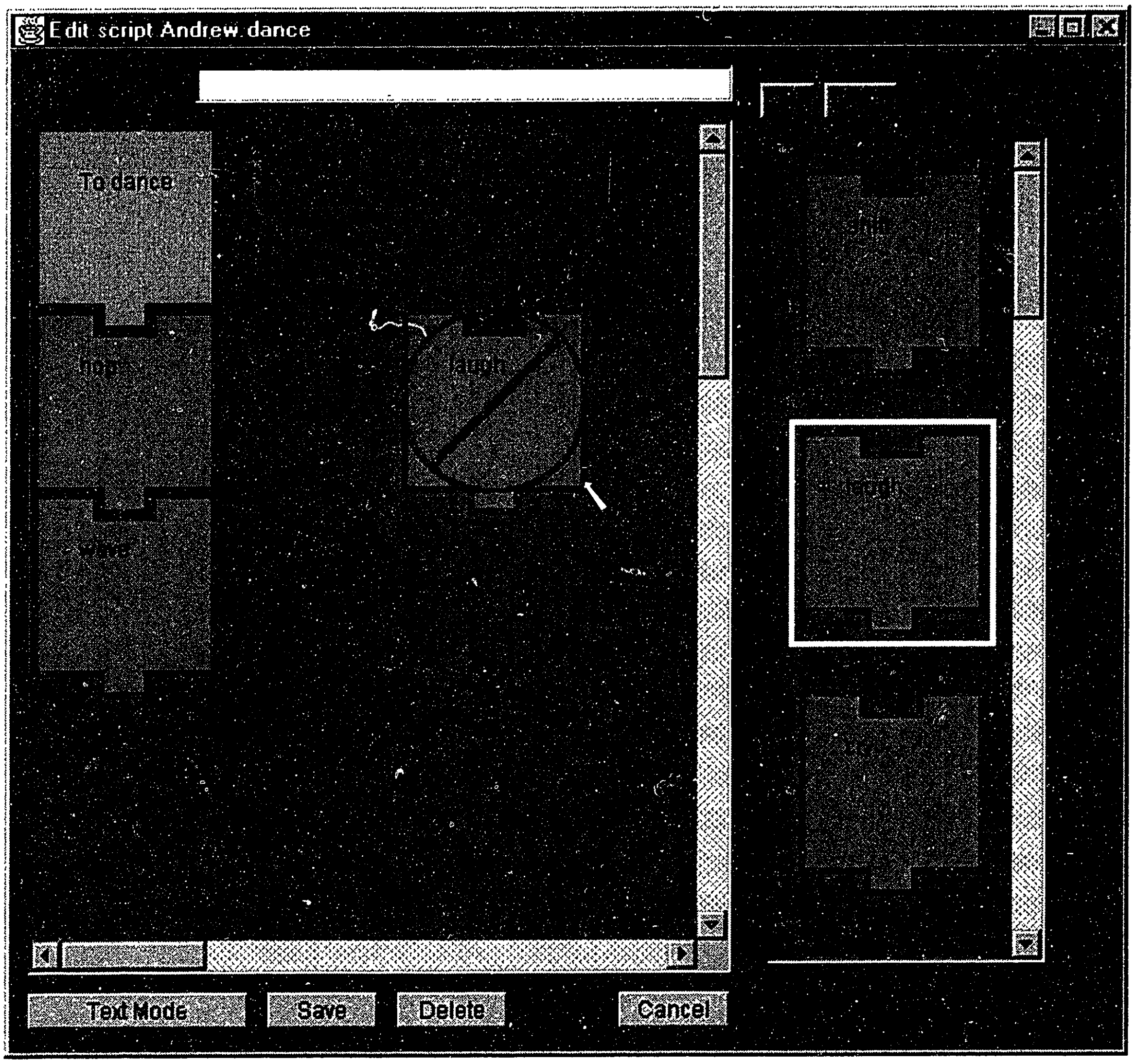


**Figure 17: Program construction in progress**

when they are connected in a vertical column with the beginning of the sequence at the top. Programmers drag a block from the palette and drop it below the block they want to connect it to; the blocks snap together if they are compatible. Figure 17 shows that a programmer selected a `Laugh` animation block and is dragging a copy towards the place where he or she wants to connect it to the program. As you can see, when a block is selected in the palette, it is outlined in white to reflect that selection. The copy of `Laugh` that is being dragged within the workspace is overlaid with a large red icon denoting that the block will not attach correctly to anything if it is dropped in the current location.

Control flow statements such as `repeat` and conditional statements such as `if` are available on a separate palette. Boolean operators as well as boolean variables that are based on the properties of the avatar and the avatar's environment are also available on their own palette.

## 4.2 Types

Young programmers can learn about types from visual cues such as block shapes and colors. If two blocks do not fit together like jigsaw puzzle pieces, then they are of incompatible types or kinds. For example, the system will not allow an and block to hold an animation block as one of its arguments; the and piece only takes boolean pieces, this can be seen clearly by the corresponding shapes of the and piece and any boolean piece. Figure 18 shows an animation block, an and piece and a boolean variable piece. Figure 19 shows how the and piece fits with the boolean piece.

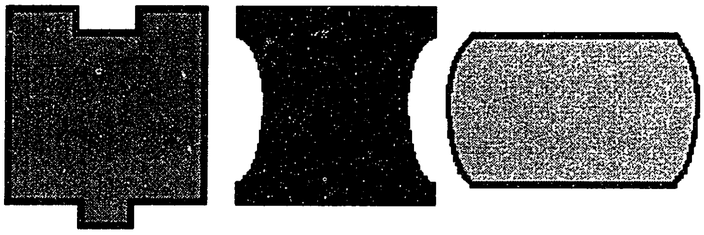


**Figure 18: An animation block, an and block and a boolean variable block**

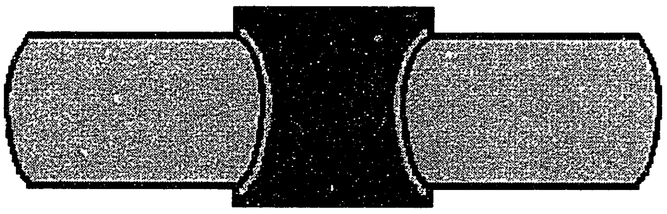


**Figure 19: A boolean operator between two boolean variables**

A color scheme further assists the programmer: dark blue boolean operators always take light blue boolean pieces. Furthermore, any side of a piece that can be connected to any other piece is shaded in the color that it expects. A dark blue boolean operator has two light blue boolean edges. Figure 19 shows the shading.

### 4.3 Overlaid Images for Feedback

If the child tries to connect an animation block to a boolean operator, the animation block is overlaid with an image which indicates that the block cannot be connected there (see Figures 10 and 17). This image is a visual cue that gives the child feedback. This feedback is very important; if it were absent, then the child might think that the system is malfunctioning when he or she drops the animation block and it fails to connect as intended. When the feedback is given, the child (1) gets the message that the block cannot connect there and (2) the system knows that and is not malfunctioning.

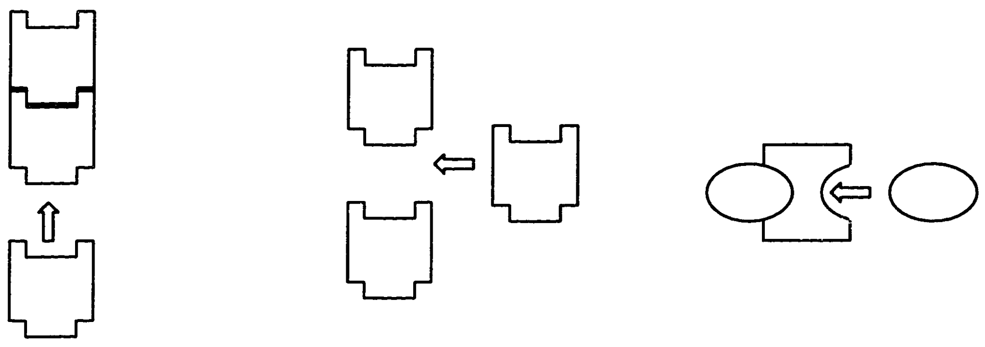


**Figure 20: Icons for (a) attaching a block to the bottom (b) inserting a block (c) attaching a block to the right end**

If the child tries to make a valid connection between two blocks, then the system overlays an appropriate image to indicate what kind of connection will take place if the block is dropped where the mouse currently is. Figure 20a shows the image that is overlaid if the current block being dragged by the mouse is to be connected at the bottom of a vertical sequence of blocks such as a sequence of animations. Figure 20b shows the image that is overlaid in the block is to be inserted somewhere within the vertical sequence. Figure 20c shows the image that is overlaid if the block is to be connected to the end of a horizontal sequence of blocks such as a sequence of boolean variables and boolean operators.[1]

## 4.4 Animations as Labels

An advanced visual cue that was discussed previously in this thesis can be used to label animation blocks. The animation block can have, on its face, a small window that shows the animation playing continuously in a loop. Therefore, the animation block is easily identified by both the animation loop and the text label. See Figure 21.[2]

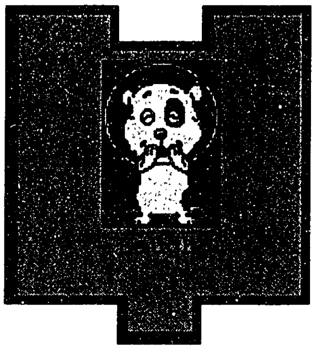


**Figure 21: Animation as label**

This advanced visual cue increases the ease with which programs can be communicated to other children. Recall that communication between members of the community is an important part of constructionist learning.

What if the animation loop was displayed without the text label? If the animation loop were complicated and long, it might be hard to identify the animation block at a glance. A programmer might have to watch the better part of the animation loop in order to know what it is. A text label identifies an animation block at the single glance. This is a trade-off that we can discuss with the children who participate in Pet Park. This kind of

[1] These particular overlaid images are not currently implemented due to time constraints. However the implementation would be trivial.
[2] This capability is not currently implemented. However it would be simple to adapt the code that is already used to display the animation in Pet Park.

conversation can be fruitful for them because they can learn how to think about slightly more complicated design issues.

## 4.5 Switching from Graphic Mode to Text Mode

The ability to switch from graphic mode to text mode is perhaps the most effective way that Pet Park Blocks helps beginner programmers transition smoothly to programming in a textual environment. The user interface for the graphical programming environment includes a button for switching to the text mode. This is a useful learning tool for a beginner programmer who wants to learn the syntax for YoYo, because if the child wants to see what YoYo syntax would look like for a new construct such as an `if` statement, he or she can select the control structures palette, drag an empty if statement into the workspace, connect some boolean properties and operators and some animation blocks, and finally click on the `switch` button to see what the text equivalent is.

However, since the graphic implementation does not cover all of YoYo exhaustively, this method of example generation will not work for all YoYo constructs. The programmer learns here that there are limits to graphical programming. This limit will encourage programmers who are mature enough, to move on to the textual interface, because they will want more expressive power. In this way, Pet Park Blocks is an addition to Pet Park that helps the environment accommodate users at whatever level they need.

## 4.6 Graphical Representation Reflects Text Equivalent

One final feature of Pet Park Blocks that could be implemented is that the graphical representation for program constructs such as an `if` statement can reflect the YoYo equivalent. For example, the graphical `if` program block can have text labels such as brackets that reflect the YoYo syntax. When the user switches from graphic mode to the text mode, the graphics can fade slowly and leave behind only the text.

# 5. Implementation: Under the Hood of Pet Park Blocks

This section describes the implementation of Pet Park Blocks. The key to understanding Pet Park Blocks is understanding the Piece, the data structure used to represent the different parts of the program that is constructed within the Workspace. The overall design of the system is shown in Figure 22. The user manipulates the Workspace and the

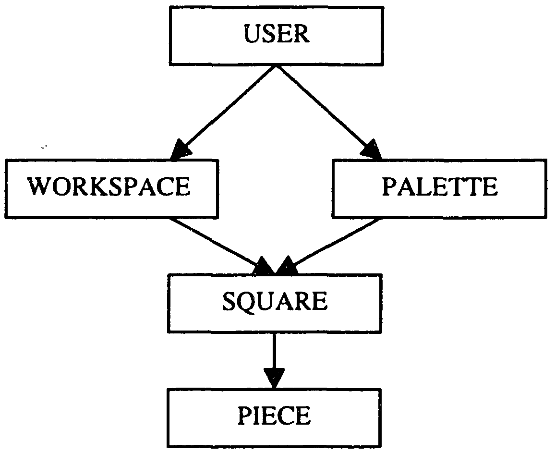


**Figure 22: Architecture of Pet Park Blocks**

Palette. Within these two spaces, Squares receive mouse events and in turn manipulate Pieces. Each Piece represents one part of the program being constructed. A more detailed description of these parts of the Pet Park Blocks system is included in the sections below.

## 5.1 Program Representation: Pieces

A Piece is a data structure that represents program parts such as animation blocks, boolean operators and variables, and control flow statements such as the `if` statement. A Piece is aware of its own type. For example, an animation Piece is aware that it can only accept connections at its top edge and its bottom edge. The Piece also contains the actual YoYo code that it represents. When Pieces are connected together, there exists a total

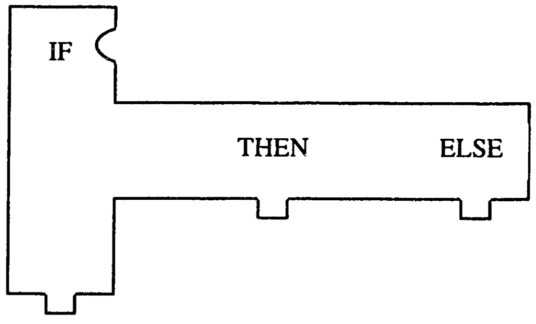


**Figure 23: An `if-else` program block**

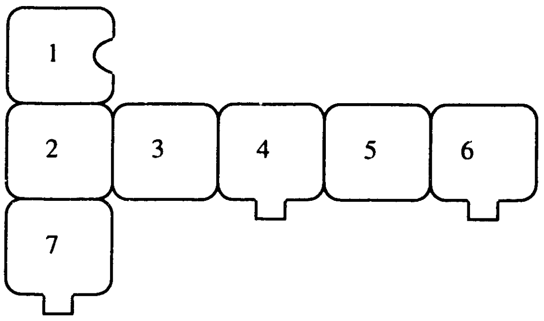


**Figure 24: Total ordering of pieces**

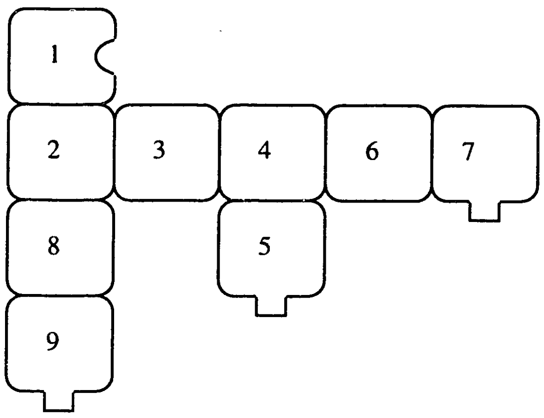


**Figure 25: Expansion unit (8) added to accommodate for piece attached to (4)**

ordering, because each Piece has links to the previous Piece and the next Piece. Pieces are aware of what other Pieces are connected at each of the four sides. Figure 23 shows an `if-else` statement. Figure 24 shows the Pieces that comprise the statement and the total ordering of those Pieces. The Pieces in Figure 23 that have no text label are expansion units, which are automatically added to accommodate for whatever is connected to the `then` and the `else` Pieces. Units 3 and 5 are expansion units. Figure 25 emphasizes the expansion unit added to accommodate the Piece was connected to the `then` Piece.

The total ordering is used when the YoYo code is required. The first piece of the program passes its code to the `next` piece, which appends its own code to the code it receives. It then sends this new accumulated code to its `next` piece. This continues until the last piece in the program has all of the code.

## 5.2 Graphical Representation of Pieces: Squares

All Pieces are represented graphically by Squares, which are basically Java Canvases. Squares can accept certain mouse events depending on whether they reside in the Palette space or in the Workspace. Mouse events determine what commands a Square sends to its Piece.

## 5.3 Palettes

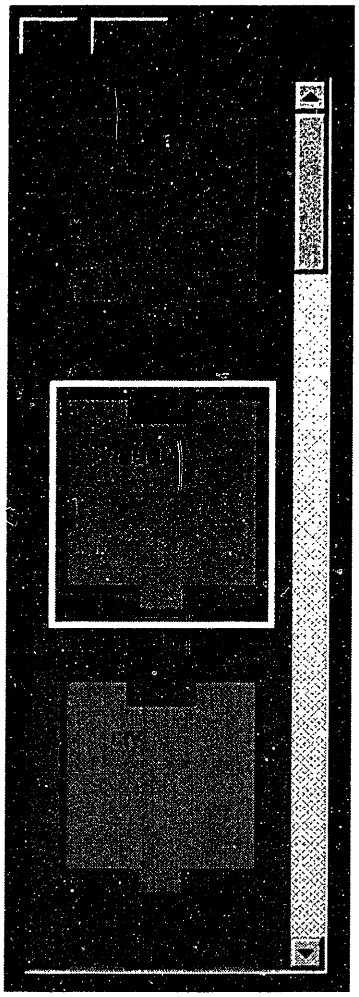


Figure 26: A palette

Palettes are the libraries that hold the program blocks that the programmer can use when constructing a new program. See Figure 26 for the Palette that holds the animation blocks for Spotnik. A Palette is basically a scroll pane with an array of Squares. When a Square receives a Mouse Down event, it changes its image to indicate that its Piece has been selected and that it has been copied into memory. The Mouse Down event also clears the memory of old Pieces and un-selects any Pieces that were previously selected. The Square outlined in white in Figure 26 is a selected Square.

## 5.4 Workspace

The Workspace is a two-dimensional matrix of Squares, which receive mouse events such as Mouse Down, Mouse Entered, and Mouse Exited. See Table 1 for a description of what happens in each case.

| Mouse event | Piece selected | No piece selected |
|---|---|---|
| Mouse entered | Square displays appropriate visual cues to indicate whether selected piece can connect to program at this point. | No action. |
| Mouse exited | Square reverts back to normal appearance. | No action |
| Mouse down | If selected piece can connect to program at this point, Square attaches it to the program and the system updates the rendering of the program. | If the Square is not empty, its piece is marked for deletion. Otherwise, the Square takes no action. |

**Table 1: Mouse events handled by the Workspace**

# 6. Future Work and Concluding Remarks

## 6.1 Finish Implementing Visual Cues and Other Tools

There are still a few very useful visual cues that are not yet implemented. For example, each Square is labeled with a text label and a static image. We can include the animation itself as a label, as described in Section 4.4.

Another visual cue that would be useful is animation of parts of Pet Park Blocks. For example, when the programmer drags a block into the Workspace and attaches it to the program, Pet Park Blocks can animate the act of actually connecting the two blocks and also play a clicking sound when the two blocks touch. If the programmer tries to make an illegal connection, for example dropping an animation block next to a boolean operator, the system can animate the two program blocks colliding a few times to indicate that they do not fit together.

Another tool that can be added to the system is a `Try` button which, when pressed, causes a small window to appear with the current program running. This way the programmers can stop occasionally and see what they have created.

## 6.2 Encapsulate Programming Rules in Knowledge Base

The programming rules are currently embedded within the Piece. The Piece knows which other kinds of Pieces it can connect to, and on which of the four sides, etc. This knowledge is hard to alter once it is embedded. A good improvement to the system would be to encapsulate the knowledge about such programming rules in a knowledge base. This knowledge base would then be consulted whenever the system wants to check whether a certain action is legal. This way, the knowledge would be easy to alter and extend, apart from the actual Java code of the Pet Park Blocks system.

Along with the knowledge base, we can include the capability to give informative error messages. For example, after any illegal operation, the system can print out the reasons why the action was illegal. This error message is based entirely on the inferences that the system made using the knowledge base. The explanation capability is common in knowledge-based systems and it lends to the transparency of the task at hand. Another way to provide this feature is to show the messages only when the programmer clicks on a button marked `Why was that illegal?`

## 6.3 Include Other Programming Paradigms

Pet Park Blocks can be expanded to include other programming paradigms. For example, to write automated movement behaviors for an avatar, Pet Park Blocks can offer programming by example, much the way Cocoa does (see Section 3.3). We can ask the children about the different ways of programming. They may form their own ideas about the different programming paradigms.

## 6.4 Empirical Studies

The system has not yet been tested thoroughly to see its effectiveness in providing scaffolding for beginner programmers. Formal experiments should be conducted to determine to what extent Pet Park Blocks lowers the cognitive threshold for young beginners. Do some of the younger children find programming in the original textual environment daunting? Austina De Bonte, the designer of Pet Park, has seen children as young as 7 programming in YoYo with relatively few problems. Are there children who cannot program in YoYo, but who can operate in a graphical programming environment? These are questions that can only be answered with formal empirical studies.

Also studies should be conducted to gather feedback from children. We would like to know what features the children would like to add to Pet Park Blocks and which current features and visual cues are awkward or unclear. It is our hope that Pet Park can be adapted for classroom use so that it enhances the opportunities for learning about programming and communication.

## Acknowledgments

First, I am constrained to thank God for teaching me what I really needed to know and for giving me the grace to make it through MIT. Second, I would like to thank my parents and my brother and sister for being such a great family. I would also like to thank Alicia and Chris and all the good friends in God's family. I would repeat the struggle of MIT just to be with all of you again.

My loudest thanks goes to (1) Mitchel Resnick, my advisor, for his patience and grace, (2) Austina De Bonte for her excellent guidance, and (3) Jaime A. Meritt for his hard work which gave me inspiration.

## References

[1] Albright, Michael J. & Graf, David L., Editors. *Teaching in the Information Age: the Role of Educational Technology.* Jossey-Bass Publishers. San Francisco. 1992.

[2] Apple Computer, Inc. "Apple's Cocoa Site." http://cocoa.apple.com/cocoa/home.HTML

[3] Begel, Andrew. "LogoBlocks: A Graphical Programming Language for Interacting With the World." MIT, 1996.

[4] Begel, Andrew."Bongo Home." http://el.www.media.mit.edu/bongo/

[5] Bigge, Morris L. *Learning Theories for Teachers.* Harper & Row. New York. 1964.

[6] De Bonte, Austina. " Pet Park: a Graphical, Virtual World for Kids." Master's Thesis Proposal, MIT, 1997.

[7] De Bonte, Austina, " Pet Park Home." http://www.media.mit.edu/~austina/petpark.html

[8] Bruckman, Amy. "MOOSE Crossing: Construction, Community, and Learning in a NetworkEd Virtual World for Kids." Ph.D. Thesis, MIT, 1997.

[9] Cooper, James M. & Ryan, Kevin. *Those Who Can, Teach.* Houghton Mifflin Company. Boston. 1995.

[10] Epistemology and Learning Group, MIT Media Laboratory. "A Cricket: Tiny Computers for Big Ideas." http://el.www.media.mit.edu/people/fredm/projects/cricket/

[11] Gardner, Howard. *The Unschooled Mind: How Children Think & How Schools Should Teach.* BasicBooks. New York, NY. 1991.

[12] Holt, John. *How Children Learn.* Pitman Publishing Corp. New York. 1967.

[13] Lewis, R. & Tagg, E.D., Editors. *Trends in Computer Assisted Education.* Blackwell Scientific Publications. London. 1987.

[14] Papert, Seymour. *Mindstorms: Children, Computers, and Powerful Ideas.* BasicBooks. New York, NY. 1993.

[15] Papert, Seymour. "Uses of Technology to Enhance Education." MIT A.I. Laboratory, June, 1973.

[16] Repenning, Alex. "Agentsheets & VisualAgenTalk HomePage." http://www.agentsheets.com/

[17] Resnick, Mitchel. *Turtles, Termites, and Traffic Jams.* MIT Press. Cambridge, MA. 1993.

# THESIS PROCESSING SLIP

FIXED FIELD: ill. ______ name ______

index ______ biblio ______

► COPIES: [✓ Archives] Aero Dewey [✓ Eng] Hum

Lindgren Music Rotch Science

TITLE VARIES: ► ☐

NAME VARIES: ► ☒ ~~[illegible]~~ Chun-Ho

~~[illegible]~~ middle name

IMPRINT: (COPYRIGHT)

► COLLATION: 41 p

► ADD: DEGREE: ______ ► DEPT.: ______

SUPERVISORS:

NOTES:

cat'r: date:

page:

► J138

► DEPT: E.E.

► YEAR: 1998 ► DEGREE: M.Eng.

► NAME: CHENG, Andrew C.