Broadcast-Based Communication

in a Programming Environment for Novices

by

David S. Feinberg

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 20, 2004

Copyright 2004 M.I.T. All rights reserved.

Author_____
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by_____
Mitchel Resnick
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 20 2004
LIBRARIES

BARKER

Broadcast-Based Communication

in a Programming Environment for Novices

by

David S. Feinberg

Submitted to the

Department of Electrical Engineering and Computer Science

May 20, 2004

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis proposes a model to make communication and processes in an object-oriented language accessible to novice programmers. For my thesis work, I developed a broadcast communication framework in the context of Scratch, an object-based programming language intended for children and being developed at the MIT Media Laboratory. Objects in this framework can broadcast events, which trigger other objects to start new processes. In this document, I identify the basic functionality required to support object communication in such a broadcast model, I describe my implementation of that functionality, and I consider the merits of various user interface metaphors to make these features accessible to novices. Finally, I compare the event broadcasting in this model with the use of procedures in more traditional programming languages, and raise questions as to whether the broadcast model's event handlers should supplement or supplant procedures in a beginner's programming language.

Thesis Supervisor: Mitchel Resnick

Title: Director of the Lifelong Kindergarten group at the MIT Media Laboratory

ACKNOWLEDGMENTS

There are many people I would like to thank for their contributions to my thesis:

- Eric Grimson, for making MIT affordable, by giving me the opportunity to TA for 6.001.
- The Media Lab's LLK group, especially those on designed Scratch--without which it would have been immeasurably more difficult to add support for broadcasting events in Scratch.
- The Scratch UROPs, for their hard work and contagious enthusiasm.
- Brian Silverman, whose unique perspectives provided the roots for much of my thesis work.
- My thesis supervisor, Mitchel Resnick, who arranged funding and a quiet office for me, trusted me to work on the Scratch language, always made time for me, kept my best interests and career in mind, suffered through reading this document, and signed it.
- My mentor, John Maloney, who taught me everything I know about Squeak, and created much of the original Scratch architecture that became the starting point for my work. Our Seattle's Best conversations inspired most of the ideas in this document.
- My unofficial thesis expert, Margarita Dekoli, who invaded my office, provided invaluable suggestions for organizing my thesis, shared crossword puzzles with me, let me snack in front of her, and became a good friend--even if she is made of coffee.
- Eytan Adar, who listened to my thesis troubles and filled me with ABP discount pastries.

I also wish to thank all the students, friends, and family members that have helped me survive two unbearably cold Boston winters, and made it so difficult to leave again. I am especially grateful to my roommates--Jonathan, who conceals his boredom as I babble about my day, and Jen, who has been entertaining and ever supportive in our nighttime chats.

I would also like to thank my future roommates, Jackie and Mr. Moustache, who have been storing up three semesters of craziness for my return. Most of all, I would like to express my love to Virginia, their owner, who has allowed and encouraged me to go back to school, spent all her

vacation days with me, made sure I graduated, let me come home again to her, and loved me through it all.

Finally, I would like to thank myself for actually writing this thesis. "I made this."

CONTENTS

PREFACE

Computer programming can be a powerful medium for children to express their creative ideas. Every day, URL links send us to entertaining animations and games created by young authors. Nearly all such projects involve graphics representing cartoon characters, animals, vehicles, etc. In other words, these projects manipulate graphical objects. The most natural way to bring such objects to life is through object-oriented programming, in which the program's code mirrors the graphical object whose behavior is being described.

If we wish to develop an object-oriented language to empower children and novices to create such animations and games, then we must consider ways to help young programmers to learn advanced programming topics. One topic at the core of object-oriented programming is the *method call*--the means by which an object communicates with others in an object-oriented system. In a typical method call, the calling object waits while another object performs the requested computation. However, in an interactive story or game, there are typically several graphical objects, all simultaneously following their own rules. This means that the corresponding programming objects are each running one or more processes. Writing programs to choreograph the communication between such objects involves the starting and stopping of processes, and the intricacies of inter-process communication.

This thesis presents an intriguing communication model that takes steps toward making these advanced programming concepts accessible to novice programmers. In this model, objects broadcast events to trigger other objects to start running new processes. This thesis shows how such a mechanism can be made accessible to young programmers, and how it can be used to create the many kinds of behaviors required for the kinds of projects that will be of interest to children.

This document is primarily divided into two parts. In the first part, consisting of chapters 1-6, I present the broadcast communication model that is the focus of my work. This work was developed in the context of Scratch, a programming language intended for children and being

developed at the MIT Media Laboratory. Chapter 1 presents some background information on the Scratch project, and chapter 2 describes how objects communicated in an early version of the Scratch programming language. Chapter 3 then presents the basic functionality of the broadcast communication model, and chapter 4 proposes various user interface metaphors to make this model more accessible to novice programmers. Chapter 5 explains how this model was implemented, and chapter 6 explores a simple game developed using broadcast communication.

The second part of my thesis, chapters 7-11, pushes the broadcast model much further. I consider this part to be a response to the first. Chapter 7 describes how my simple game demonstrated a need to allow programmers to invent names for broadcasted events. By using string names for events, however, I encountered a significant tension--the broadcast functionality I added began to resemble the procedure-call functionality already supported by Scratch. Although these mechanisms were designed to solve different problems, their largely overlapping functionality would be a point of confusion for Scratch users. Therefore, chapter 8 describes how the broadcast model can be used to simulate procedure calls, and chapter 9 considers the merits of replacing the procedure-call mechanism altogether. In Chapter 10, I summarize the key questions raised by my thesis work, and present my answers to those questions. Finally, chapter 11 describes my suggestions for future work in using a broadcast model for object communication in a programming language for novices.

I. BACKGROUND

The Value of Computer Programming

Research has shown that children learn best through creative explorations in which they design, create, and invent things they care about [Resnick, Rusk, & Cooke, 1998]. Seymour Papert coined the term "Constructionism" for this educational philosophy [Papert, 1993]. It suggests that we can aid in a child's learning by providing tools and media that are engaging in addition to being educational, and that support a wide range of creative possibilities, opening new opportunities for children to create and to reflect on their creations.

Computers are such an educational medium because they offer ways to create, design, and invent--especially through the use of programming languages. Children can use programming as a means to explore complex creative ideas by making artwork, presenting stories, developing games, etc. As Papert observed in his book "Mindstorms: Children, Computers, and Powerful Ideas", the computer can play a number of educational roles [Papert, 1980]. In programming computers, children learn to "speak mathematics", using arithmetic and geometry in order to reach the authentic goals they set for themselves. Programming also serves as a concrete introduction to formal and abstract reasoning. And, in telling the computer what to do, children learn to think formally about their own thinking.

Some people feel that you do not truly understand an idea until you can teach it. Programming provides endless opportunities for a child to "teach" their ideas to the computer. And in correcting program errors, children learn the powerful metacognitive skill of debugging their own thinking [diSessa, 2000; Kay, 1991].

The Logo programming language was created by Papert and his colleagues for children and other novices. The language features an on-screen object called a "turtle" that the programmer controls in order to create interesting shapes and patterns. By imagining themselves moving along the turtle's path, children are able to learn abstract geometric concepts [Papert, 1980]. Although once

met with widespread excitement, interest in teaching children to program has waned over the years. This decline reflects the limited success that educators have had in introducing children to programming. Some researchers feel that such attempts to teach children to program have focused too much on programming languages that are difficult to learn and that do not easily support activities that connect with children's interests.

Lessons from the Computer Clubhouse

One environment where children can use the computer as a creative medium is provided by the Computer Clubhouse. Started in 1993, the Computer Clubhouse is an international network of after-school centers in under-privileged areas, where kids between the ages of 10 and 18 can go to engage in computer-related activities in their free time [Resnick, Rusk, & Cooke, 1998]. One of the most successful activities has been the creation of computer artwork. In fact, a kind of "Photoshop culture" has emerged, with members actively exchanging how-to tips and sharing their creations. Unfortunately, computer programming has had only scattered success in the Clubhouses, and no such "programming culture" has developed [Resnick, Kafai, & Maeda, 2003]. Only a limited number of members have created projects using Flash or Shockwave, and some have been exposed to MicroWorlds Logo.

One possible obstacle preventing a programming culture from emerging could be the difficulty of learning a programming language--perhaps because of complicated syntax rules or inappropriate primitives. Another hypothesis is that those programming languages that are easier for children to learn (those with a "low floor", such as Logo) have not allowed children to create projects they find engaging. These beliefs have led to the conception of the Scratch programming language [Resnick, Kafai, & Maeda, 2003].

Introducing Scratch

The goal of the Scratch project [Resnick, Kafai, & Maeda, 2003] is to develop a graphical programming language that is easy to learn and will connect with children's interests, particularly through the use of multimedia and sensory input. It is hoped that Computer Clubhouse members (and hopefully many others) will be excited to use Scratch to create Flash-like animations, games, etc.

Scratch features moving objects that support various multimedia, including images, movies, and sounds. Children will be able to manipulate images by programming their own filters, build sensor devices to direct movie playback, and control object movements from mouse input. Scratch brings together ideas from three successful programming languages in the educational technology community--MicroWorlds Logo, EToys, and LogoBlocks. The Scratch programming language (especially its primitives) is based on Logo, and features an environment like MicroWorlds in which multiple objects (similar to turtles) can move around and interact with each other.

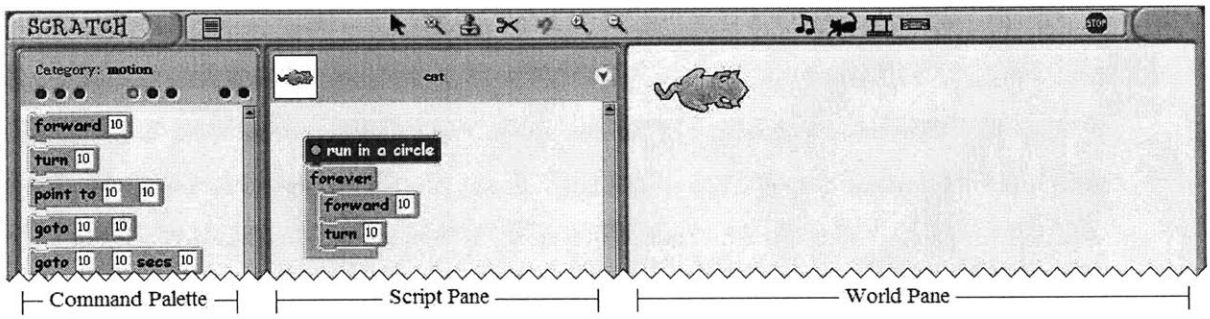
Like EToys [Steinmetz, 2001], Scratch is developed in Squeak (an implementation of SmallTalk) [Guzdial, 2001], and hence shares many features with EToys (such as creating an object automatically when the user draws a picture). Also like EToys, Scratch is programmed by dragging and dropping blocks (instead of typing). However, the way in which the blocks snap together to form programs more closely resembles the Lego-inspired LogoBlocks software [Begel, 1996]. Unlike its predecessors, Scratch is strongly integrated with various multimedia. The intention is to encourage Computer Clubhouse kids to create multimedia projects similar to those they have developed using applications like PhotoShop, but with the added power of programmability.

One significant departure from Logo concerns the connection between objects (or turtles) and the programs associated with them. Logo is a procedural language, in which the programmer adds procedures to a single code repository. These procedures issue instructions to choreograph the behaviors of the various objects. On the other hand, in using an object-oriented programming

language, the programmer focuses on the data (objects), and the procedures detailing the behaviors associated with a particular object are built directly into that object's declaration.

The Scratch Programming Environment

The following graphic shows the upper portion of a screenshot from the March 16, 2004 image of Scratch.



Scratch Screen Layout

The screen is divided into three main regions, which I will refer to as the *command palette*, *script pane*, and *world pane*. The world pane is the stage upon which the characters interact. The programs that govern those interactions appear as *scripts* (also sometimes referred to as *procedures*) on the script pane, and the blocks that make up these scripts are dragged out from the command palette. These three regions appear on the screen at all times, rather than popping up in dialogs. This way, the key aspects of the Scratch development environment are visible to the novice user from the beginning.

In the project shown, the user has created a script called "run in a circle", that tells the "cat" object to (1) move forward a distance of 10 pixels, (2) turn right by 10 degrees, and continue to run these two commands "forever". We refer to the top block, labeled "run in a circle", as a *script hat*, used to give a name to the script's behavior. Invoking the "run in a circle" procedure causes the cat object to trace out a circular path in the world pane. During this time, a green indicator appears

on the script to indicate that it is currently running. Scripts and script hats will play a prominent role in this document, because the object communication models we will be considering are intimately connected to how scripts are invoked.

Scratch is an interactive language like Lisp or Squeak, in which there is no clear separation between compile-time and runtime. The programmer frequently makes changes to a script while others continue to run. In fact, Scratch exhibits a deeper "liveness" than Lisp or Squeak, because the programmer does not create classes to be instantiated later. Rather, the programmer is always conjuring up and manipulating live objects.

Scratch Objects

Although the work for this thesis pertains to Scratch objects associated with various types of multimedia, I will restrict my examples to simple objects that appear as an image moving and interacting in the world pane (like the cat object shown earlier). As in other object-oriented languages, a Scratch program consists of a number of objects. However, in an effort to make programming easier for novices, Scratch does not feature certain characteristics of other object-oriented languages like C++ and Java. First, the objects in Scratch can contain procedures (scripts) and data (a number of fields with changing values), but they cannot contain other objects. More significantly, programming in Scratch does not involve classes or their associated complexities. Every object is the sole instance of the code describing it. (Objects can be copied, but the "instances" are not connected in any way.) As a result, Scratch does not support user-defined types, inheritance, full polymorphism, and other object-oriented features related to classes.

A key aspect of Scratch toward fostering a programming culture is that objects created by one programmer can be shared with others. Hence, we imagine that a child may find a dog object in a library, and modify it to be able to follow its owner around. Then the child could export the "following dog", and another kid could use it as part of their video game.

Although support for sharing Scratch objects had not been implemented (as of this writing), it is helpful to imagine the sharing of a Scratch object as involving taking a live Scratch object (one that appears in the world pane and may be associated with running scripts) from one project and moving an unborn copy of it (one with no processes running) into a library of objects. The copy has fields and scripts that are identical to the original. Its field values match those of the original at the time the copy was made. (It is possible that some field values should be modified when the object is exported to the library. For example, x- and y-coordinates may no longer be meaningful. Furthermore, it may not make sense to retain the values of any fields that refer to other objects.)

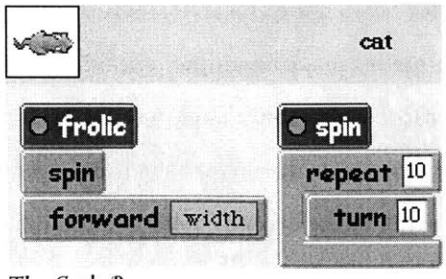
II. COMMUNICATION IN SCRATCH 0.1

One of the core challenges we faced in developing Scratch concerned how objects communicate with each other and with themselves, through the calling of procedures and starting of processes. Before we can discuss the broadcast communication model that is the focus of this thesis, it is helpful to understand the earlier Scratch communication model that led to it.

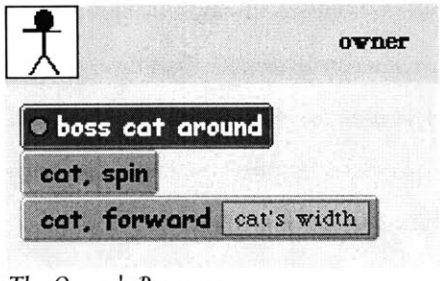
In 2003, we developed a novel communication model that allowed any Scratch object's script to invoke any other object's primitive commands and scripts. Furthermore, when invoking a script, the programmer could choose to start it in a new process. I will refer to this 2003 version as *Scratch 0.1*, to distinguish it from the current version of Scratch and from the hypothetical versions considered in this document.

Hard-Wired Blocks

A core feature of the Scratch 0.1 communication model is that nearly all program blocks contain hard-wired references to the object they influence. This feature allowed the user to program one object to influence the behavior of a second object simply by including blocks from that second object inside the first object's scripts. Look at the following sample programs:



The Cat's Program



The Owner's Program

The cat object's program shown was created by dragging blocks from the cat's command palette into its script pane. Concealed in each of these blocks is a reference to the cat object. When the

cat's "frolic" script is run, the "spin" block knows to call the cat's spin script, then the "forward" block knows which object to move forward, and the width variable knows which object's width to report. The owner object's program above was created by placing the same blocks (from the cat's command palette) inside the owner's script (in the owner's script pane). Because of the potential for confusion with hard-wired blocks, Scratch labels each block explicitly with the name of the cat object. Running the owner's script causes the cat to behave exactly as if the "frolic" script were run.

In the Scratch user interface, when you want to look inside one of the objects in the world pane, you double-click it, prompting the command palette to show blocks relevant to that object, and the script pane to show that object's scripts. Therefore, the command palette and script pane typically show elements for the same object. However, in order to drag the cat's "forward" block (for example) into the owner's script, the user must request to view the cat's blocks in the command palette and the owner's scripts in the script pane at the same time.

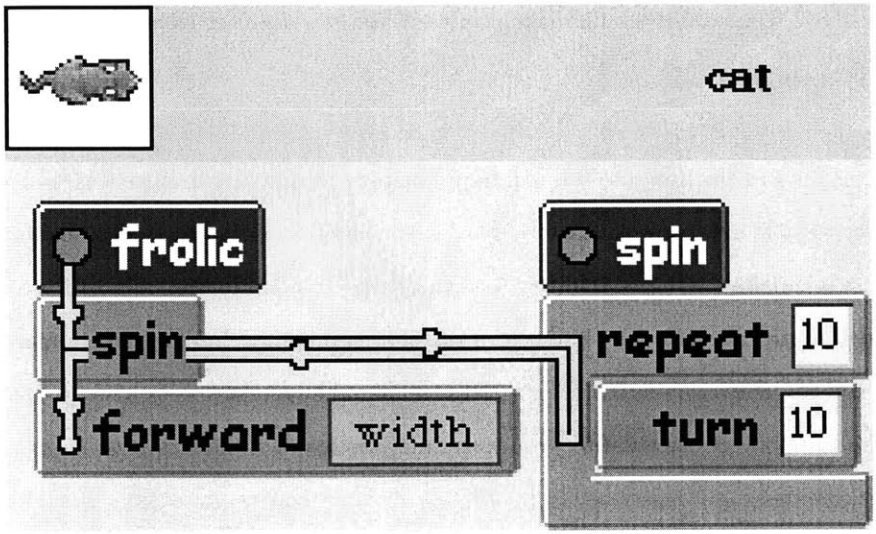
Unfortunately, it is unlikely that the user would think to do this, or that there should be an obvious user interface gesture to initiate the situation. And, should the user succeed in finding this gesture, the display will enter the confusing state of showing details for two different objects at the same time. This scenario represents a typical programming need in Scratch, and is therefore cause for some concern with the hard-wired block model.

A larger concern involves shareability. A basic requirement for Scratch is to be able to share any object across projects. This means that no object can require the presence of another specific object in order to function. In early user testing, we found that the Scratch 0.1 communication model had the unfortunate effect of encouraging users to create projects in which the various objects' scripts heavily cross-referenced each other. In our example, the owner object could not be moved into a project without the cat object, since the blocks in the owner's script contain hard-wired references to the cat. It would make little sense to share the owner or any object that is similarly hard-wired to potentially absent objects. We must therefore consider alternative inter-

object communication models, in which an object is not granted full access to another object's behaviors.

Explicitly Calling Procedures and Starting Processes

We will now take a closer at the procedure call used in the cat's program. The yellow arrows in the following diagram have been added to show the flow of control when the "frolic" script is run:



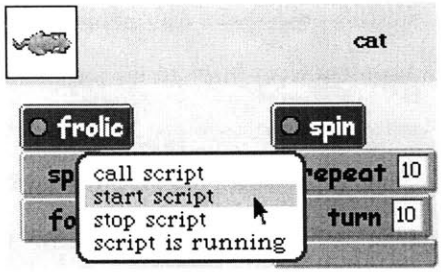
Control Flow for a Procedure Call

First, the indicator on the "frolic" script hat turns green, to show that there is now a process associated with this hat. Next, the "spin" block (representing a call to the "spin" procedure) is reached, and control abruptly jumps to the spin script. When execution of that script has completed, control returns to the calling script ("frolic") and the "forward" block is executed. Finally, having reached the last block in the script, Scratch turns off the "frolic" hat's indicator light, and execution terminates. Notice that, in the course of these events, the indicator on the "spin" script never turns green.

We also wanted one process to be able to cause a procedure (like "spin") to start running in a new process. Then, the new process would run in parallel (simultaneously) with the remainder of the original one. Although our experience in developing programming languages had shown that the ability to run multiple processes was a powerful and important concept even for a beginning programming language, we were concerned that the idea of processes would be potentially too advanced and elusive for novice programmers. We did not want a Scratch user to face the complexity of processes as they appear in traditional object-oriented languages, in which the programmer creates, runs, and keeps track of processes by identification numbers or references to special process objects. In such languages, several processes can be running the same script simultaneously--a feature that would make it difficult to show the user what processes are currently running in Scratch.

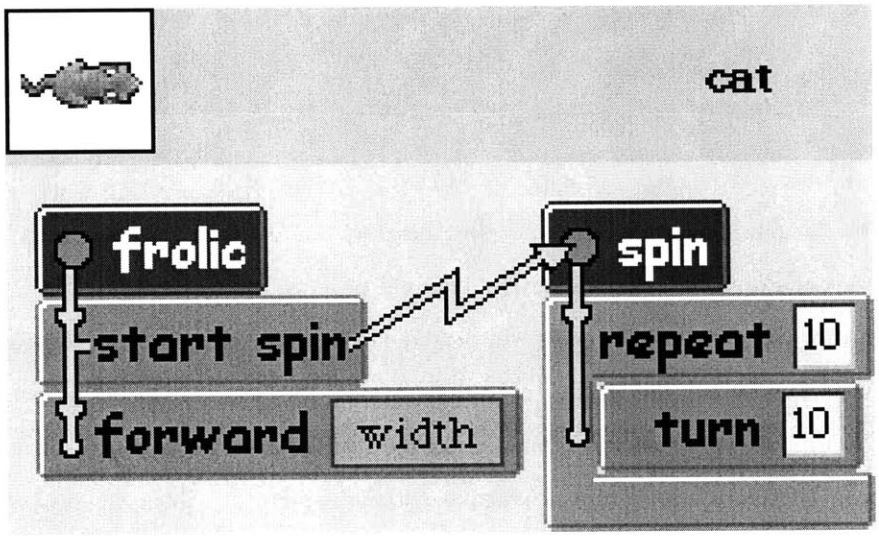
We therefore decided that the simplest solution would be to associate each process with a script hat, and not to allow any hat to have more than one process. This way, the script hat can indicate if the process is running, and the script's name can be used to identify the process unambiguously inside a program (in case we want to stop it or test if it is running). In the unlikely case that you needed to have an arbitrary number of processes running the same script, you could make many copies of the same object, each with a process running one copy of the script.

In the Scratch 0.1 procedure call example earlier, there is exactly one process involved, and therefore there is exactly one script hat ("frolic") whose indicator has turned green. In order to program the "frolic" script to start the "spin" script in a new process, the user grabs a "spin" block and indicates that it should start a process, as opposed to invoking it as a procedure call. In our interface, the user right-clicks on the "spin" block, and a pull-down menu appears as shown below. The user then selects "start script", and the block's text changes to read "start spin".



Choosing to Start a Process

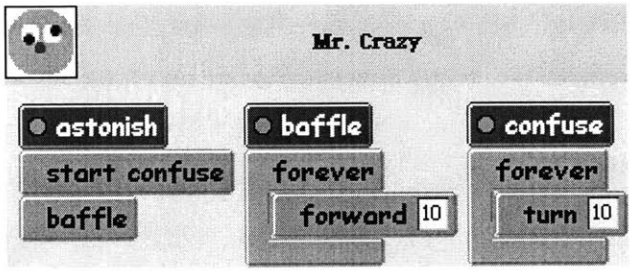
The following diagram shows the control flow when the user starts this modified "frolic" script:



Control Flow for Starting a Process

When the "frolic" script is started now, the "frolic" hat's indicator turns green to show that it is now associated with a process. This "frolic" process then reaches the "start spin" block, which forks off a new process to start the "spin" script. At this time, the "spin" hat's indicator turns green, showing that it too is now associated with its own process. From here, both processes run in parallel, with the "frolic" process continuing to run the remainder of the "frolic" script (the forward block), and the "spin" process running its "repeat" block, seemingly at the same time. As each process completes, the light on the corresponding hat turns off.

Although this explicit means of starting processes is elegant and innovative in some ways, it was ultimately felt that the model was potentially too confusing for a novice programmer. In particular, it felt surprising that a script hat's indicator could be green even when the process was running in another script (one whose script hat indicated it was not running a process). Consider Scratch's behavior when Mr. Crazy's "astonish" script is started in the program below:



Process Confusion

First, only the "astonish" process is running, and therefore the "astonish" hat's indicator is green. Then the "start confuse" block spawns a new "confuse" process, and the "confuse" hat's indicator turns green, too. The "confuse" process will continue to cause Mr. Crazy to turn forever. The "astonish" process now calls the "baffle" procedure, moving the "astonish" process's control into the "baffle" procedure (where it will remain, since the "baffle" script never returns from its forever loop). Therefore, the "astonish" script remains green, but the "astonish" process is running inside the "baffle" script, which does not turn green. This behavior stands in sharp contrast to the behavior of the "confuse" process, which remained safe in the confines of the script with the same name.

Would a novice programmer understand the reasons for this surprising and seemingly inconsistent behavior? These concerns ultimately led us to consider alternatives to starting processes in this explicit manner, which seemed too much like cloaking professional computer programming languages in drag-and-drop blocks. We began to search for a more innovative and intuitive way to introduce parallelism to novices.

III. BROADCASTING EVENTS

Alternative Communication Models

Growing concerns about the lack of shareability and confusing nature of processes in Scratch 0.1 led us to consider alternative models for object communication. One possibility would be for Scratch to take a more traditional object-oriented approach, invoking methods on variables that refer to objects, rather than hard-wiring all program blocks with references to the object they influence. In an object-oriented language like Smalltalk, the compiler will allow any message to be passed to any variable referencing an object. For example, the Smalltalk syntax "pet spin" passes the "spin" message to the object stored in the "pet" variable. Only at runtime might the message fail to invoke a method (if the object in the "pet" variable does not have a "spin" method).

Following this model, imagine a Scratch owner object whose code passes the "spin" message to its "pet" variable, which currently references the cat object (however that might appear in our graphical user interface). If the owner is shared and imported into another project, the owner code will only fail if, when run, the pet variable happens to reference an object that does not support the "spin" message. Although I did not take this path in my work, we will later see how the broadcast-based model I explored ultimately evolved into something resembling this standard message-sending model.

The communication model I considered for my thesis work is one in which objects broadcast events to each other. This model was selected because it easily supports shareable objects and because it provides an intuitive way to think about starting processes. For example, in this paradigm, the cat object may have a script that responds to blue events. Whenever a blue event is broadcasted, the cat's blue script starts running in a new process. For the owner object to tell the cat to run this script, the owner must broadcast a blue event. All objects responding to blue events, including the cat, will run any associated scripts (in new processes). In fact, a significant benefit of a broadcast paradigm is that it greatly simplifies the code required to trigger many simultaneous actions--a common usage pattern in our experience so far with Scratch programming.

Because Scratch already uses strings to name procedures, we initially chose to use colors to name events. We hoped that this choice would reduce any possible confusion between the use of procedures and events.

Artist's Depiction of Objects Broadcasting Events

Of course, one obvious drawback of this model is that now the user cannot designate a specific object to be the recipient of an event. It is unclear how significant this loss of functionality is. Although it is possible to devise scenarios where addressing a particular object would be necessary, we have so far been impressed by how much we can program in Scratch without this functionality.

Because an event serves as an intermediary between objects, our example's "owner" object no longer explicitly depends on the presence of the cat object, and can now be moved into projects that do not contain a cat. If no objects in the project respond to blue events, then the event is simply ignored. It is as if the owner said "spin" in a room full of pets that do not know how to spin. An enlightened owner should therefore not be surprised when no pets respond. The broadcast communication model here is ultimately based on our intuition as to what happens when we make an announcement in a room full of other characters, each potentially responding in a different way. The remainder of this document will describe the design, implementation, and analysis of such a broadcast-based object communication scheme in Scratch.

A particularly appealing consequence of the broadcast model is that it provides some of the benefits of class-like interfaces without explicitly incorporating classes into the Scratch language. Any object that responds to blue events is essentially an implementation of the blue-event interface. If we had 10 copies of the cat object, all would respond to the same events. In a sense, it would be as if they were all instances of the same class. We would therefore have class-like behavior in Scratch, without the conceptual complications of explicitly creating and maintaining classes.

In fact, the broadcast paradigm contains a wealth of powerful computer science ideas. Each object implementing the blue-event interface may respond to a blue event in a different manner. This means that Scratch would support a simple form of polymorphism. It also means that novice programmers will be exposed to the important distinction between an interface (the events an object listens for) and its implementation (the behavior triggered by those events). And in choosing how their objects will communicate through events, novice programmers will unknowingly be developing their own rudimentary protocols.

Event-Broadcasting Model

Initially, we will primarily be considering the problem of inter-object communication. Objects will still invoke their own scripts using explicit procedure calls, as previously explained, passing arguments and returning values. On the other hand, we will no longer support explicitly starting processes in this model. A script will therefore need to broadcast an event in order to start a process for another script in the same object. Likewise, the only way for objects to communicate with each other will be through the broadcasting of events. We will no longer support the Scratch 0.1 mechanism of invoking one object's hard-wired command block inside another's script.

So far in this document, we have referred only to user-triggered events (identified by color), but there are other kinds of events to consider. In particular, we wish to be able to write scripts that respond to certain key presses and mouse clicks. We have also considered more complex events to support triggering a script when a certain external sensor value crosses a threshold, or when two objects collide on screen. However, we feel it is sufficient for now to be able to write a script that continuously tests for such a condition and broadcasts a colored event when one occurs.

One question that will come up again later in this document concerns our decision to label user-defined events by color. Suppose we have eight unambiguous colors that users may broadcast. How often will a programmer wish to create a Scratch project requiring more than eight kinds of events? A further difficulty with colors is that it is easy to forget the significance of each color.

What did we intend by a blue event? We could avoid such problems by allowing the user to use string names for events. However, using strings might make getting started in Scratch just a little more difficult for the novice programmer, who would have to consider appropriate names and use consistent spellings. Furthermore, using strings to name both procedures and events might lead Scratch programmers to confuse calling procedures with using events to start scripts.

One intermediate option would be to have the user interface suggest colored events by default, but to allow the more advanced user to designate a string name instead. I kept in mind this possibility of later using strings for events as I began to develop the Scratch event system. This is an example of one of the principles that guided me in designing the broadcast framework: the more common tasks (like designating eight events) must be easy, and the rest (like using strings to make additional events) should be possible.

Primitives for Broadcasting and Receiving Events

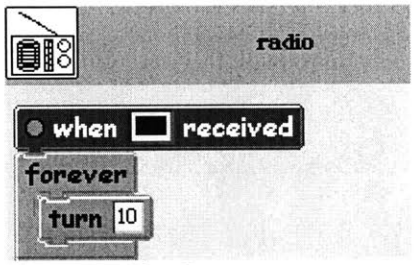
In discussing the design of the Scratch event-broadcasting system, we will first be concerned with what functionality is supported. Only later will we consider details of an appropriate graphical user interface. The block names used in this section should be considered placeholders only until we are ready to consider various metaphors for presenting the broadcast functionality to the novice programmer.

Our first new primitive is the "broadcast event" block, which a script can use to broadcast a colored event. When the programmer clicks on the event icon, a pull-down menu appears for selecting a color to broadcast, as shown.

Selecting a Colored Event to Broadcast

To program an object to respond to an event, the user associates a script with it, using a new kind of "when event received" script hat, and selects the appropriate color. (Although I considered allowing the same hat to respond to more than one type of event, this ultimately did not seem necessary.) For now, we will leave the old script hat available for creating privately scoped procedures, but there will be more said about the two types of script hats later in this thesis. The following pictures show a program in which the radio station object broadcasts a blue event to trigger the radio object to start spinning:

Broadcasting a Blue Event

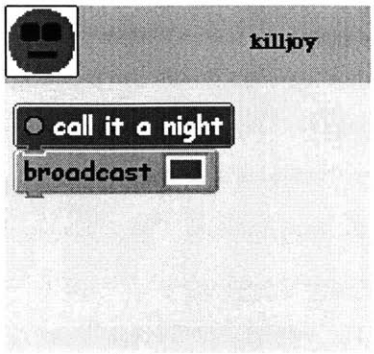


Receiving a Blue Event

Scratch's "forever" block (which appears in the radio object's script above) has been a popular instruction block. It repeatedly executes a sequence of blocks until the associated process is terminated. Typically, a process is terminated when the user explicitly clicks either (1) on the corresponding script hat, or (2) on the stop-all button on the Scratch tool bar. One question we must now address is how such a process could be stopped programmatically. A possible solution would involve associating both a start and stop event with each script. This way, broadcasting an event could stop the process associated with a script, but this solution seems somewhat cumbersome. Another solution might be to "unbroadcast" an event. In this design, "unbroadcasting" a blue event would stop any processes that were originally started by a blue

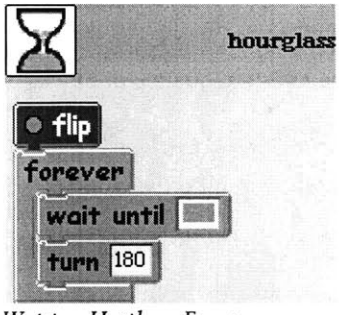
event. This is an intriguing design that should be seriously considered in future Scratch development. However, for my thesis work, I chose to add a more general-purpose "loop until event" block, and it has played a central role in my event-broadcasting model. This block, shown below, repeatedly executes a sequence of contained blocks until a designated event occurs.

Partying Until a Red Event



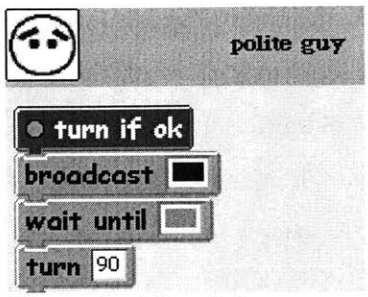
Stopping the Party

In the example above, "fun-loving guy" continues to move in a circular path until "killjoy" (or any other object) sends a red event. This event terminates the loop, and the associated process then proceeds to evaluate the "set heading" block that follows. If a "loop until event" block contains no blocks at all, it can be used simply to wait until a particular event is broadcasted. Because this behavior has been rather useful, I added a separate "wait until event" primitive block. In the example below, the hourglass object uses this block to wait repeatedly for an orange event, turn 180 degrees when one is received, and then wait again.

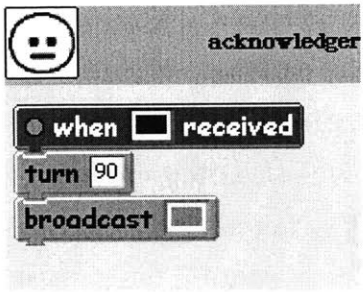


Waiting Until an Event

There is some possibility for confusion between the "when event received" script hat and the "wait until event" block. And because both blocks can be used in similar ways, the user must sometimes choose which control structure is more appropriate for a given problem. One situation in which the "wait until event" block is particularly useful is for receiving an acknowledgment from an earlier broadcast, as pictured below:



Waiting for Acknowledgment



Broadcasting a Response

In this program, "polite guy" broadcasts a blue event and then waits for acknowledgment. Upon receiving the blue event, "acknowledger" turns 90 degrees and broadcasts a green event to indicate the turn is finished. Having now received the green acknowledgment event, "polite guy" goes ahead and turns, too.

Communicating Data Between Objects

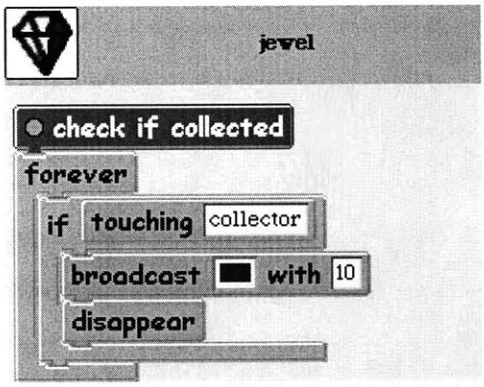
The situation is more complicated if we wish to pass data between objects, rather than simply notifying them that an event has occurred. For example, suppose we want to make a game, in which a human-controlled player moves around the screen collecting various jewel objects, each worth different point values. When the player collides with a jewel, the player needs to know its point value (perhaps because the value influences the player's behavior).

We can take two approaches to address this problem. The simplest involves using global variables. When the jewel collides with the player, it could update a global variable with its value. The player would then query this variable. The problem with this approach is that it is fairly limited

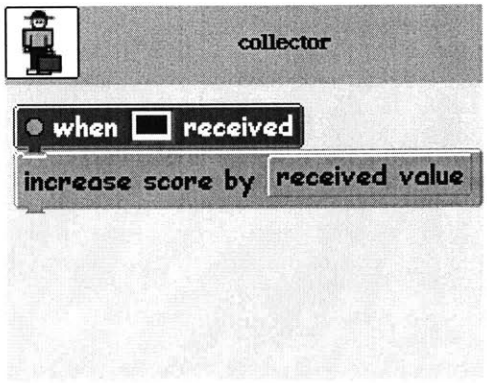
(does not generalize to many characters collecting jewels) and leads to possible race conditions (for example, if the player reaches a second jewel before reading the value from the first collision).

For my thesis work, I settled on a somewhat more elegant approach, in which objects can broadcast an event containing an argument value. In this design, a new "broadcast event with value" block now lets the programmer send a single numeric argument (or any value returned by a reporter block). A "received value" block reports the argument value from the most recently received event, which may be either (1) the event that triggered the script to start, or (2) the event that terminated a "loop until event" or "wait until event" block. (We also considered replacing the "broadcast event with value" block by using a new "set event value" block and the original "broadcast event" block in conjunction. However, this seemed unnecessarily complex.)

In the example program below, when a jewel collides with the collector, it uses a new "broadcast event with value" primitive block to send a blue event (for example) containing the jewel's point value. The collector, upon receiving the event, uses a new primitive "received value" block to pull the point value out of the event.



Broadcasting an Event with an Argument Value

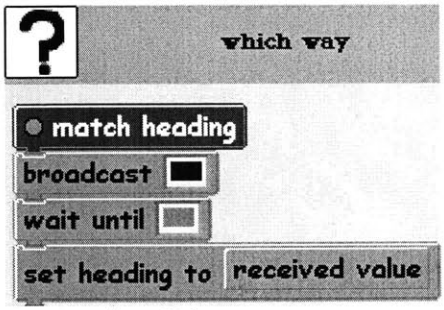


Using an Event's Argument Value

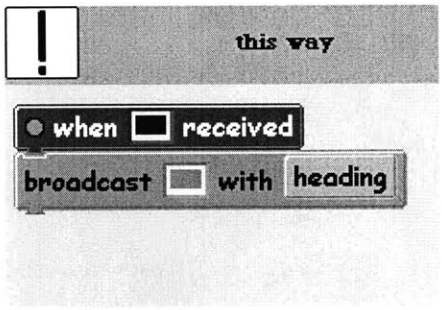
Although passing data with an event is a fairly clean solution, it does complicate the simpler case of broadcasting an event without an argument, if only because it means additional primitive blocks in the command palette to distract the novice user from the more commonly used ones.

One concern is whether it will be sufficient to have a single numeric argument value packaged with an event. It may later be necessary to support arguments of different types, or multiple arguments in the same event. (If arrays were added to Scratch, we could pass a single array value with an event, and store multiple argument values in the array.) It may also later help to provide a primitive block that would return a reference to the object that sent the event.

We can also use event arguments (in a roundabout way) to have one object query another for a value. In the following example, the "which way" object queries the "this way" object for its heading by sending a blue event, and then waiting for a green response. The "this way" object responds by sending back a green event containing its heading. The "which way" object then uses the "received value" block to access this heading information.



Querying for a Value



Responding to a Query

IV. METAPHORS

A primary reason for introducing a broadcast-based communication architecture was to make processes more accessible to novice programmers. We would therefore be doing a great disservice to our users if we simply added the new primitive blocks exactly as described so far. Instead, we should seek ways to make our event model usable and understandable, by connecting it to our intuitions of how people and devices communicate with each other in the real world. In this chapter, we will explore various metaphors we can use to represent broadcast communication in Scratch. This will mainly affect the text labels on the blocks and choice of icons. More significantly, though, a metaphor can strongly influence the way a novice programmer thinks about communication in Scratch.

Searching for an Appropriate Metaphor

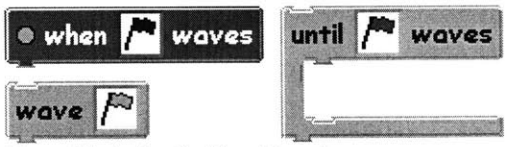
In choosing a metaphor, we will consider how to name our events. So far, we have used colors as event names, but colors are not things that we are used to broadcasting. Typically, we hear about "broadcasting" in the context of radio and television signals. Perhaps a Scratch object could broadcast a transmission on a certain radio frequency, and other objects could be tuned to listen to such radio stations. However, radio and television signals send data continuously, and our objects broadcast signals only occasionally. Therefore, our communication model more closely resembles the way one broadcasts an instruction to a radio-controlled car. This would be a sensible metaphor, but it is probably too obscure for most potential users. Few people have seriously considered how their radios and remote-control cars work, and choosing a numeric frequency as an event name would be too abstract.

A more tangible metaphor involves communicating through noises, such as drums or animal sounds. For example, we could broadcast an event with a "hit cymbal" block, and respond with a "when cymbal heard" script hat. Such a sonic metaphor would require icons (or text) representing each of the possible noises that serve as event names. However, since our program appears visually

in the form of command blocks, we would prefer instead to select a visual communication metaphor.

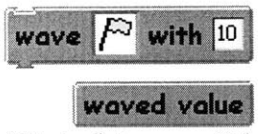
The Flag Metaphor

Another possibility involves using colored signals. We considered light bulbs and signal flares, but eventually settled on colored flags. In this metaphor, an event is broadcasted by a "wave flag" block, which may trigger a script with a "when flag waves" hat, or terminate an "until flag waves" loop, as pictured below:



Event Blocks for the Flag Metaphor

Because flags are a visual metaphor, in which we think of objects seeing flags waving, we could even designate a place on the screen to show the programmer an appropriately colored flag waving whenever an event is broadcasted. Another nice feature of the flag metaphor is that many programmers will be familiar with situations where colored flags really do signal the start of events, such as in a car race, or in nautical communication. One drawback of using flags is that a real flag would need to be in a real person's line of site in order to trigger a response. However, the real failing of the flag metaphor arises when we try to send arguments with events. What does it mean to wave a flag with a value?



"Waving" Argument Values

The Envelope Metaphor

Because we felt that passing argument values was necessary for the Scratch event system to be sufficiently powerful, we worked to find a metaphor we could safely use to pass arguments. We considered using colored envelopes to represent events, because it makes perfect sense to place an argument value inside an envelope. In this metaphor, we use a "send envelope to all" block to broadcast an event, we start scripts with a "when envelope received" script hat, and we terminate loops with an "until envelope received" block, as pictured here:

Event Blocks for the Envelope Metaphor

Like all metaphors we have discussed, the envelope metaphor was imperfect. Real envelopes are addressed to a recipient, but we need envelopes to be "broadcast" to all Scratch objects. To achieve this behavior with a real mailing, it is necessary to prepare an envelope for each recipient. Because the broadcasting concept is at the core of our communication model, can we allow ourselves to represent events by a real world object that cannot truly be broadcasted? An additional difficulty with the envelope metaphor relates to the transient nature of events. If we are not home when the mail arrives, the envelopes we receive simply accumulate in our mailboxes, which we can check later at our convenience. But in our event system, if an object is not "listening" for an event (by having a "when event received" script hat or by running a "loop until event" instruction block), then that event is forever lost to the object. It is impossible for the object to query for what events it missed earlier when it was not listening. Therefore, choosing to use the envelope metaphor may potentially surprise users who come to Scratch with intuitions about how real world envelopes behave.

Sadly, it appears that we are faced with an over-constrained problem, in which there is no perfect metaphor. Each of the various metaphors we have considered has its disadvantages. Of the possible meanings to assign to colored events, my personal choice is to use flags. However, we

will revisit this topic later in this document, when we are faced by the limitations of using colors to name events.

V. IMPLEMENTATION

Note: Some readers may wish to skip these more technical implementation sections and proceed to reading about more high-level issues concerning our broadcast communication model in the next chapter.

The Scratch Evaluator

Before we can see how the broadcasting of events is implemented in Scratch, it is necessary to take a brief, high-level tour of the Scratch evaluator. This is the part of the code that brings a Scratch program to life, by walking through and activating the various Scratch commands. In presenting the architecture of the Scratch evaluator, I will omit many details, and only focus on those elements that are relevant to later understanding the implementation of our event model.

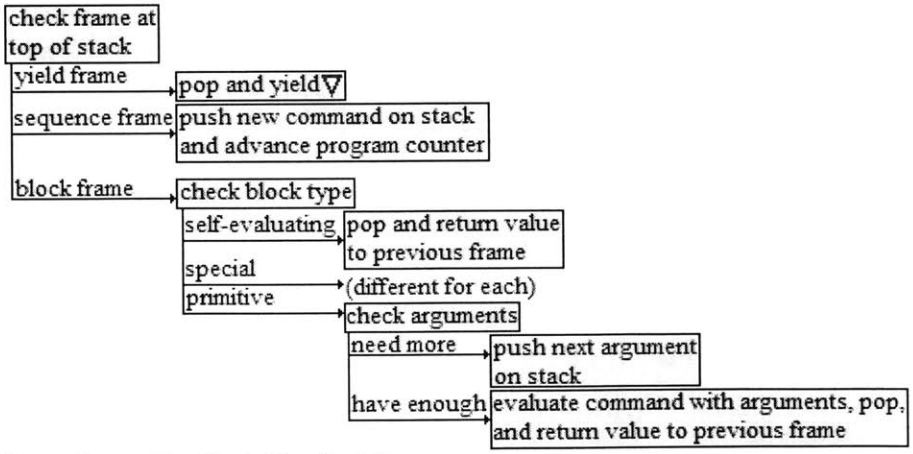
Before we get started, there are two key points to note about evaluation in Scratch. The first is that Scratch is an interpreted language. The scripts that a user writes are not compiled into any other format. Instead, the evaluator walks through and evaluates the graphical blocks themselves. The second key point is that Scratch *processes* (which run the scripts that Scratch programmers write) are not implemented in terms of Squeak processes (which are built into the programming language that Scratch is written in). In developing Scratch, we wanted to have explicit control of when to run which Scratch process, and it was therefore necessary to create our own processes and process *scheduler*.

The Scratch process scheduler maintains a list of running processes. The list grows as new processes are started, and shrinks as those processes complete or are aborted. The scheduler repeatedly runs each process in the list in order, for one *step* each. After running for a short time, each process must complete its step by voluntarily *yielding* control back to the scheduler, so that the next process may run. A process should yield control whenever it is about to perform a computation that could take a long time. Typically, this means that a process will yield at the end

of any loop iteration. (Processes also yield during timed commands like "wait 10 seconds", but such commands are not covered in this document.)

A Scratch process frequently yields control in mid-computation, only to resume later. Because a Scratch process may yield at any time, it must always keep track of exactly what it has computed so far and what it has left to do. This state information is stored in the process's *stack*, which functions as a sort of to-do list. The stack contains a pile of *frames*, each of which describes some action to be taken by the evaluator. There are three types of frames that appear on the stack: (1) a *block frame*, with a single Scratch block to be evaluated, (2) a *sequence frame*, with a sequence of command blocks to be evaluated, and (3) a *yield frame*, used to indicate that the process is ready to yield control.

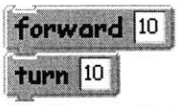
When a Scratch process runs for a step, it checks the frame at the top of its stack, and takes some corresponding action. Having taken action, there will now be a new frame for the process to handle at the top of its stack. This repeats until the process encounters a yield frame (prompting it to yield to another process) or runs out of frames in its stack (causing it to terminate altogether). The following high-level schematic shows the decisions used by a process to handle the frame at the top of its stack:



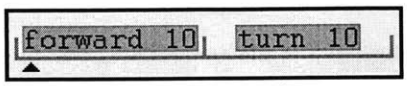
How a Process Handles Its Top Stack Frame

Evaluation of a Simple Program

The evaluator is best explained by walking through an example. Consider the following short program:



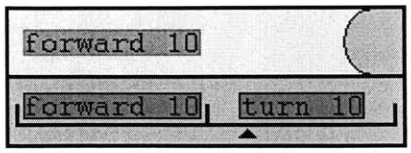
The Program Being Evaluated



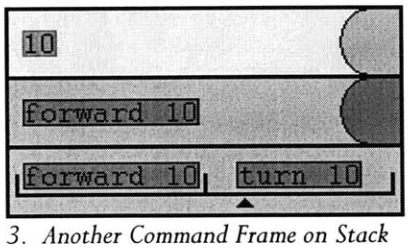
1. Sequence Frame on Stack

This program is run inside a Scratch process in a single time step (without yielding control to another process). The process is initially started with a stack containing a single sequence frame, as shown above in stage (1). This frame contains a list of the two commands to evaluate, along with a *program counter* that indicates which one to evaluate next. Initially, the program counter points to the forward command, since that is the one that must be evaluated first.

When the scheduler allows the Scratch process to run for one step, the process looks at the top of the stack and sees the sequence frame. It then advances the program counter to point to the next command in the sequence, and pushes a new block frame onto the stack for evaluating the forward command, as shown below in stage (2):



2. Command Frame Pushed on Stack

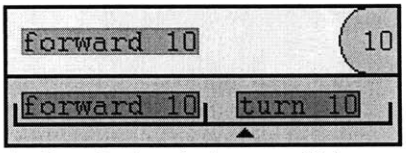


3. Another Command Frame on Stack

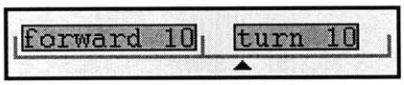
Next, the process sees the block frame on top of the stack, and its behavior will now depend on what type of block it must evaluate. There are three types of blocks: *self-evaluating blocks* (such as numbers and booleans, whose values are trivial to compute), *primitive* (the majority of blocks fall into this category), and *special* (ones with special evaluation rules like *forever*, *repeat*, *if*, etc). The

"forward" block is a primitive command. To evaluate it, we first compare the number of arguments it requires (one) with the number of argument values we have so far computed (zero, as designated by the empty bubble on the right side of the block frame). This means we must first push a new block frame containing the "10" argument on the top of the stack, as shown in stage (3).

This time, when the process sees the new block frame on top of the stack, it recognizes the "10" block to be self-evaluating. It therefore asks the block to evaluate itself, and thus the value *10* is computed. The "10" block frame is now popped off the stack, and the computed value *10* is placed in the forward block frame's list of argument values. This value appears in the bubble on the right side of the forward block frame in stage (4) below:



4. Pops Back with Argument Value

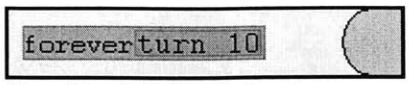


5. Ready for Next Block in Sequence

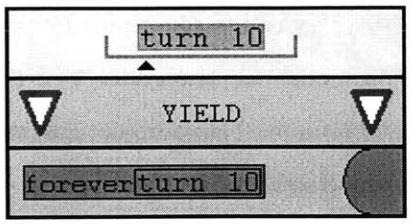
This time, the process will see that it has all the argument values it needs to evaluate the forward block, and will now ask the block for a method name. This method ("forward:") is invoked on the Scratch object in question with the argument values (10), prompting the object to move forward in the world pane by 10 units. We then pop the block frame off the stack, and we are ready to evaluate the next block in the sequence, with the stack as shown in stage (5).

Evaluating a Forever Block

We are now ready to understand how a special block like "forever" is evaluated. (The behavior of a forever block is the basis for implementing the "loop until event" blocks described later.) The following pictures show the before and after states of the top of the stack, when a "forever [turn 10]" block is evaluated.



Top of Stack Before Evaluation



Top of Stack After Evaluation

Initially, the process has a block frame at the top of its stack. This frame contains our forever block. In evaluating this block, the process notices that the forever command is a member of a short list of blocks with special evaluation rules. The rule for a forever block tells the process to add two additional frame to the stack--a yield frame and, on top of that, a sequence frame for evaluating the list of blocks contained by the forever loop. (In our example, the list consists of a single turn block.) The process will first evaluate and pop off the sequence frame. Then, having evaluated the blocks inside the forever loop, the process will pop off the yield frame and yield control, allowing the scheduler to run all other processes for one step each. When control returns to this process, the stack will be in the same state it started in, with the forever block's frame at the top of the stack. Thus, the loop will repeat in this manner indefinitely.

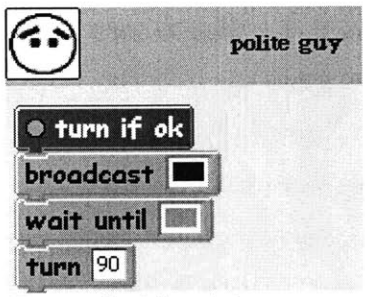
Broadcasting Events and Starting Processes

Note: To separate issues of implementation from user interface and metaphor issues, I will revert to our earlier raw, metaphor-free language when discussing the implementation of the event system.

To support the broadcast communication model, an event argument was added to the user interface. These arguments appear in the various event-related blocks in the form of colored icons (of flags or envelopes). When a process encounters an event argument, it returns a string representing the event name ("blue", for example) as the argument's value. Support for naming events with arbitrary strings was implemented deliberately, in anticipation of later allowing users to enter their own event names.

The "broadcast event" command is implemented as a primitive block, meaning that it does not require a special evaluation rule. Its event argument is evaluated normally, and the block simply leverages primitive behavior built into the Scratch object that is doing the broadcasting. This primitive behavior causes Scratch to create a new event object and broadcasts it. This event object contains the event's name ("blue", for example). (Later, the event will also contain an argument value, and could one day include information about the object that broadcasted the event.) To broadcast the event, Scratch must find all "when event received" scripts listening for events matching the name of the newly created event. For each such script, Scratch creates a new process. This process is added to the scheduler, with its stack initially containing a single sequence frame with the list of blocks appearing under the script hat.

One open issue concerns what to do when an event is sent to a script that is already associated with a running process. We feel that a hat should never be associated with two processes at the same time, but this still allows for several possible solutions. One is to use the event as a trigger to *stop* a process that is already running. However, such behavior can be somewhat surprising to the user. Another possibility would be simply to ignore the new event. This is the behavior I ultimately implemented for my thesis work. I have found it to be reasonable for simple uses of events, but potentially dangerous when used in conjunction with blocks that wait for an event. For example, consider the "polite guy" and "acknowledger" scripts, reproduced here.



Waiting for Acknowledgment

Broadcasting a Response

If the "acknowledger" script is already running when "polite guy" broadcasts the blue event, Scratch will ignore the event, and "polite guy" will be waiting forever for a green acknowledgment event. This is a common difficulty in parallel programming when there is a shared resource (the

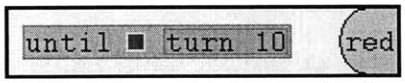
"acknowledger" script). The typical solution would be to have the "broadcast event" command wait (block) until the "acknowledger" script is finished running an earlier process and ready to start a new one. However, such a solution does not seem appropriate for a broadcast model, and has its own share of difficulties.

Waiting for Events

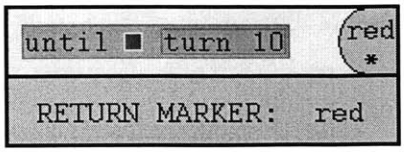
The "wait until event" block behaves exactly like a "loop until event" block containing an empty sequence of commands. We will therefore focus only on the implementation of the "loop until event" block. It was originally unclear to us how a "loop until event" block should behave. We considered treating it like a while loop, in which we only test at the beginning of each loop iteration if the designated event has arrived. But, if the sequence of commands inside the loop took a long time to execute (for example, if it included a "wait 600 seconds" block), this would mean granting an extended existence to events. We would need to keep track of what events had been received during the loop iteration, and all of this would conflict with our intention that events should be fleeting and soon-forgotten occurrences. We ultimately settled on a behavior in which an event could interrupt a process at any time.

The "loop until event" command is implemented as a special block, which means that it requires the Scratch process to apply a special evaluation rule for it. The "loop until event" block should act exactly like a forever block until an event with the designated name is received, at which point the process should continue evaluating subsequent blocks. Therefore, the evaluation rule for this block should cause the process (1) to take note of what event name it is waiting for and where to return when that event is received, and (2) to otherwise behave like a forever loop. (We will also require a mechanism for breaking out of the loop when the event is received.)

The first diagram below shows the stack as it appears after the event argument has been evaluated for an "until red received [turn 10]" instruction:



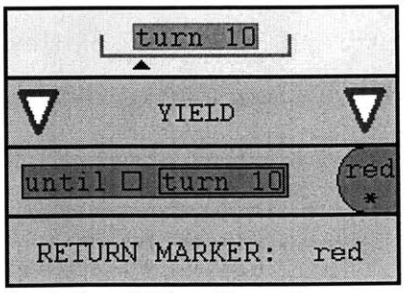
1. Block Frame with Argument Evaluated



2. Return Marker Frame Inserted

At this point, Scratch inserts a new kind of frame, called a *return marker*, as shown in the second diagram. In our example, the return marker frame tells the process that, when a red event is received, the process should (1) pop the return marker and all frames above it off the stack, and (2) continue execution at whatever frame appeared below the return marker. Until that event arrives, the process will behave like a forever loop. However, we want this return marker to be inserted only once--not on every iteration of the loop. We therefore add an extra dummy argument value (indicated by an asterisk in the diagrams) to the "loop until event" block frame, indicating that we have already added a return marker. (We could just look to see if there is a return marker already present, but what if a marker was put there for some other reason?)

On each iteration, the process will check for the dummy value. If none is found, then a return marker is inserted below and a dummy value is added to the block frame. If the dummy value is there already, the process will add a sequence frame on top of a yield frame (shown below in the third diagram), just as it did for the forever loop.



3. Evaluated Like a Forever Loop

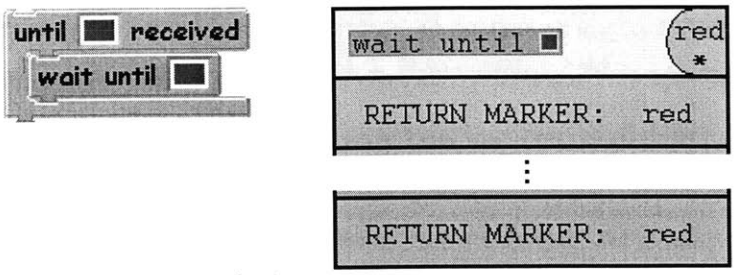
Interrupting a Loop with an Event

In order to interrupt this loop, someone must broadcast a red event. Previously, we said that broadcasting an event only prompts Scratch to start new processes (for corresponding script hats). But, this is insufficient to support "loop until event" blocks, because we now require broadcasting an event to change the behavior of processes that have already been started. Therefore, when an event is broadcasted, we will now pass the event to each existing process, in addition to starting new ones.

Recall that the scheduler only truly runs one process at a time, with all others in a sort of suspended animation. A process will only check for newly broadcasted events when the scheduler asks it to run for a step. But, this means that it is possible for more than one event to arrive before a process can examine any. When this happens, it is important that the process handle them in the order they were received. (Otherwise, nested "loop until event" blocks will exhibit surprising behavior.) Therefore, in addition to having a stack, it is now necessary for a process to maintain an *event queue*. When an event is broadcasted, it is added to the end of each process's event queue. Every time a process is about to look at the frame on top of its stack, it will now first check its event queue. (If we only checked for new events when the process is first asked to run a step, then we would miss any events broadcasted by the process itself.)

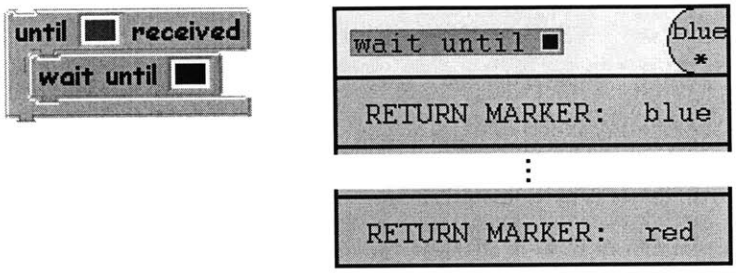
The process examines the events in the order they were received. For each one, it looks through the stack, starting from the top, for a return marker frame waiting for an event with the same name. If a match is found, that marker frame and all frames above it are popped off the stack. Otherwise, the event is ignored. One by one, each event is removed from the queue. When there are none left to examine, evaluation proceeds as usual from the top of the stack.

The details of this architecture primarily have implications for the behavior of nested loops. In the following example, we have a "wait until event" block nested inside a "loop until event" block. (Recall that "wait until event" blocks behave exactly like empty "loop until event" blocks.) Both blocks are waiting for a red event, so two red return markers appear on the stack.



Nested Blocks Waiting for the Same Event

When a red event appears in the event queue, the process will only pop the top two frames off the stack--from the top return marker on up. On the other hand, in the example below, a red event will break out of both loops, popping frames off the stack from the red return marker on up--including the blue return marker.



Nested Blocks Waiting for Different Events

Consider the same example again, but this time for the case where the event queue contains both a red and a blue event (each of which must have arrived since the last time this process yielded). If the blue event arrived first, then we will pop frames off the stack from the blue return marker on up, and then from the red marker on up. On the other hand, if the red event arrived first, we will pop all frames from the red marker on up, including the blue return marker. At this point, we will still have a blue event left in the queue. If there are truly no more blue markers left on the stack, then this event will be ignored. But, if there had been a second blue return marker somewhere below the portion of the stack shown (as would be the case if the "until red received" loop were nested inside a larger "until blue received" loop), then additional frames will be popped off the stack. In other words, the order that the events are received may affect how a process behaves.

This is why it is important to keep track of the original ordering of events by storing them in a queue structure.

Events with Arguments

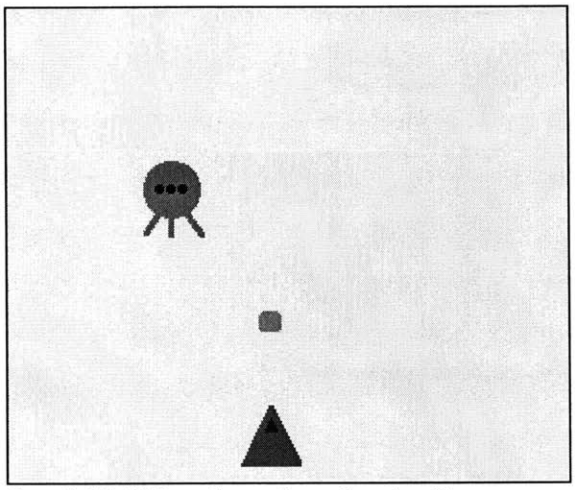
It is time to extend our broadcast model to include events with argument values. Now, when we evaluate a "broadcast blue with 10" block, Scratch will create an event object whose name is "blue" and whose argument value is *10*. When an event is broadcasted without specifying an argument (by using a "broadcast blue" block, for example), an argument value of *0* is automatically set by default. This means that a "received value" block will return *0* when an event is received with no specified argument value.

In order to evaluate a "received value" block, each process must now remember the last event it received (in addition to keeping a stack and an event queue). Therefore, when a process is created in response to an event, the process now stores this last received event. In addition, when an event received in the process's event queue results in the termination of a loop, that event gets stored as the last received event (and the previous event is forgotten). The "received block" command is implemented as a special block. To evaluate it, the process simply returns the argument value of the last received event.

VI. CASE STUDY

The Space Invaders Game

To step back and get a sense of what it is like to use events, I decided to program a simplified version of the classic "space invaders" arcade game in Scratch. Below is a screenshot of the Scratch world pane for my space invaders game:

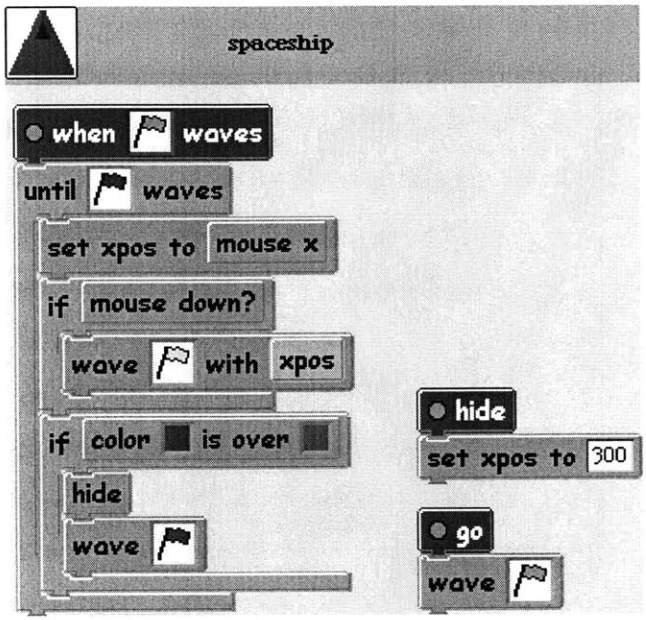


Scratch World Pane for my Space Invaders Game

My simple game consists of just three objects: (1) a triangular red spaceship that moves along the bottom of the screen as I move the mouse, (2) a small gray bullet that is fired from the spaceship when I click the mouse button, and (3) a green alien creature that zigzags down the screen. Because there is only one bullet object in the game, the spaceship can only fire one shot at a time. Once a bullet is fired, the spaceship must wait until the bullet goes off the top of the screen before firing another. If the bullet strikes the alien, the alien disappears and the game ends. Alternatively, if the alien reaches the spaceship, the spaceship disappears and the game ends.

The Spaceship

The space invaders game was implemented using blocks based on the flag metaphor we described earlier. To start the game, the user runs a special "go" script, which waves a green flag to tell both spaceship and alien that the game has begun. This script has arbitrarily been included among the spaceship's scripts, as shown below:

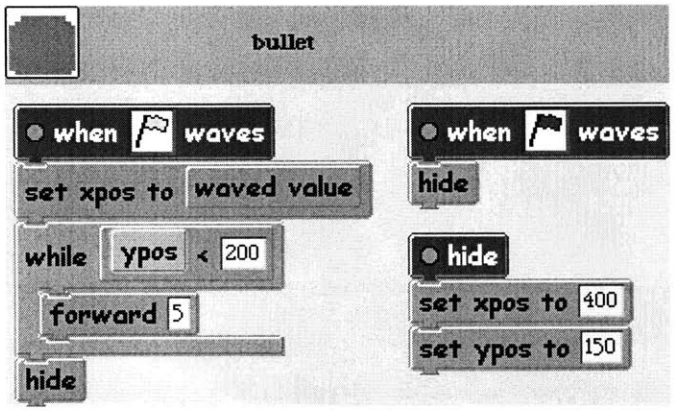


The Spaceship's Scripts

When the spaceship sees this flag, it begins running its "when green flag waves" script shown above. In this script, the spaceship repeatedly (1) moves so that its x-coordinate corresponds to the x-coordinate of the mouse, (2) fires a bullet when the mouse is clicked, and (3) dies (by hiding offscreen) and signals the end of the game if it touches the alien. To fire a bullet, the spaceship waves a yellow flag to start the bullet's firing script. Because the bullet needs to know where to start firing from, the spaceship must send its x-coordinate to the bullet as an argument with the yellow flag event.

The Bullet

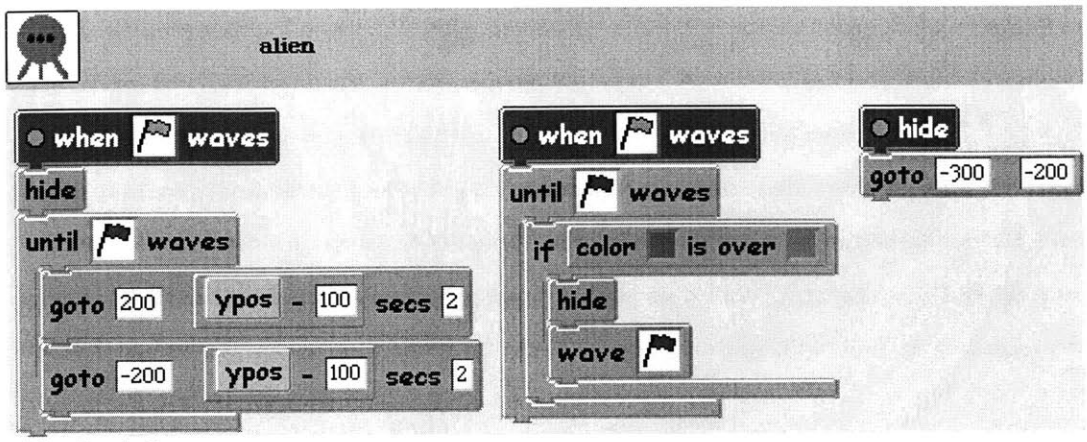
When the yellow flag is waved, the bullet fires by running the corresponding script, as shown below. It sets its x-coordinate to match the spaceship's position, as contained in the yellow flag event. The bullet then advances up the screen. The script terminates when the bullet has gone offscreen. The bullet has an additional script that causes it to move offscreen whenever a red flag waves (signaling the end of the game).



The Bullet's Scripts

The Alien

The green flag, signaling the start of the game, triggers two of the alien's scripts to start. Both of these scripts continue to run until a red flag is waved, signaling the end of the game. The first script, as shown in the following picture, causes the alien to descend from offscreen, and zigzag down the screen toward the spaceship. The second one causes the alien to die if it encounters the bullet, just as the spaceship does when it touches the alien. This behavior is realized by repeatedly testing if the alien's green color is overlapping with the bullet's gray color. If this condition occurs, then the alien dies (by hiding offscreen) and waves a red flag, signaling the end of the game.



The Alien's Scripts

Lessons from Space Invaders

The space invaders game appears to have validated the need to pass arguments in events. Initially, we hoped that events without argument values would be sufficient to generate a wealth of interesting programming projects in Scratch. However, Scratch would be far too limited if it did not allow a user to create a project as simple as my space invaders game--a project that could not have been programmed without passing values with events. Clearly, the spaceship must communicate its x-coordinate to the bullet. If all inter-object communication is to be handled through events, then packaging this value inside an event is the only reasonable solution.

There are a number of ways we could improve Scratch in order to simplify the programming effort behind my space invaders game. In my project, the spaceship's "go" script waves a green flag to start the game. But, this seems to be an unnecessary level of indirection. Rather than ask the spaceship to wave the flag, why not have the person playing the game wave this virtual flag themselves? It was therefore proposed that a button be added to the Scratch toolbar at the top of the screen. The user would press this button to wave a green flag. We are specifically choosing green here, because it is commonly associated with words like "go" and "start", and because we feel that nearly every Scratch project will require some mechanism to prompt it to start running. For the Scratch user that is beginning work on a new project, such a button must not imply that

merely pressing it will make anything actually go. The user interface must suggest to the user that the button simply broadcasts an event, and that the user may choose to have the event trigger appropriate scripts to make the project go.

One place where my space invaders scripts are unclear relates to my usage of "color is over" blocks. Instead of testing if the alien's green color overlaps with the bullet's gray color, I would just like to know if any part of the alien object is touching any part of the bullet object. Although the current version of Scratch provides a block to test if one object is touching another, this block is contrary to our plan to make Scratch objects shareable. As discussed earlier, we do not want to permit one object to contain a block with a hard-wired reference to another object. However, we can avoid this difficulty by leveraging the implicit interfaces provided by the event model. Rather than hardwiring the alien to test if it is touching the bullet, we could instead have the alien check if it is touching an object that responds to yellow events. This would require a new block, which would take in an event argument and return a boolean value. In the case of the alien testing for a collision with the bullet, it might read "touching something that responds to yellow events" or "touching something that looks for yellow flags".

A final simple improvement that could be made concerns the detection of mouse clicks. Currently, the spaceship must continuously check if the mouse is down in order to determine when the player has clicked the mouse button. It would be preferable if, instead, we could think of mouse clicks as a kind of event. In this model, Scratch would automatically broadcast a special event when the mouse is clicked, which would start any script with a "when mouse clicked" hat. Although this approach simplifies mouse handling for the programmer, it involves some tricky issues. For example, we clearly do not want all mouse clicks to fire events, since often we are using the mouse to edit a Scratch program, rather than as input to a running program.

VII. FROM COLORS TO STRINGS

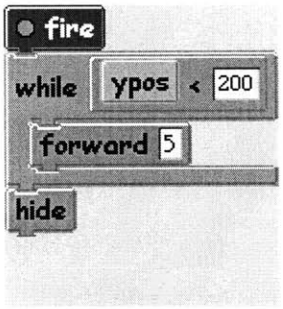
The Fall of Colored Events

The most serious lesson from my space invaders game concerns the role of colors as names for events. Recall that we chose to use colors, rather than strings, largely to avoid confusion between broadcasting events and calling procedures. However, in the course of describing the space invaders program, I find myself automatically assigning meanings to flag colors. Every time I talk about green flags, I find it necessary to remind my audience that green signifies "go". Even as I am writing this Scratch program, I am thinking "game over" in my head when I use a red flag. If you were looking at the scripts for my space invaders game for the first time, you would probably have to study the code pretty closely before you developed the intuition that yellow flags mean "fire". In developing a more complex Scratch project, researcher Margarita Dekoli found herself needing to keep track of the meaning of each color on a note written next to her computer. In fact, when describing her efforts to create a simple project with a novice programmer, Dekoli wrote, "In my explanations, I used names for the events, like 'eat you' and 'disappear'." It therefore seems that naming events with colors makes Scratch programs confusing to write and difficult for others to read, and has not eliminated the need to associate words with events--at least in the programmer's head.

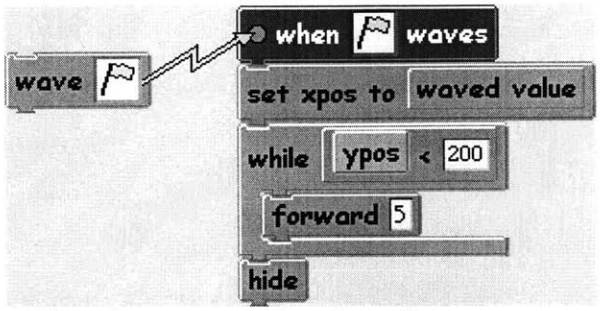
I encountered a closely related issue in writing the space invaders program, concerning script hats. With the introduction of the event mechanism, Scratch now features two kinds of script hats. First, there are the original script hats that are assigned string names and used for traditional procedure calls. We might therefore call these *procedure hats*. Now, we have also introduced *event hats* to allow the programmer to associate an event with a script. The hats used in my space invaders game are predominantly event hats, with labels such as "when yellow flag waves".

When I first began to write the space invaders program, I had no plan in mind for how I was going to leverage events. One of the first things I did was to use a procedure hat labeled "fire" to begin a script in the bullet's script pane. When I was satisfied with the bullet's firing behavior, I realized

that I needed to modify the spaceship's code to invoke the bullet's fire procedure. At this point, it became apparent that I would need to use an event here, and that I would therefore need to modify the "fire" script to use an event hat instead. I pulled off the procedure hat, deleted it, created a new event hat, and connected it to my old "fire" script. I chose to use a yellow flag as an appropriate reminder of "fire". The two versions of the "fire" script are shown here:

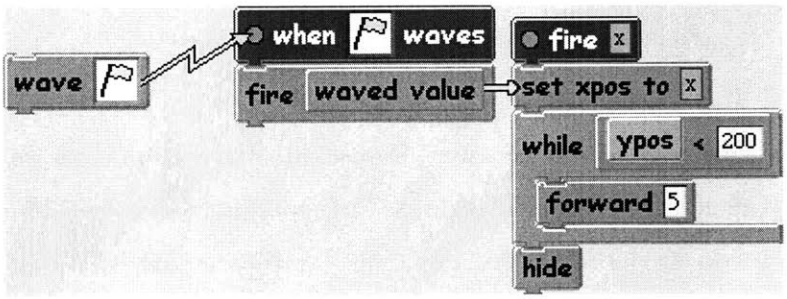


The Fire Script with a Procedure Hat



Invoking the Fire Script with an Event Hat

Now, it seems a shame that, by virtue of associating the script with an event, I should have to give up a descriptive name like "fire". It was therefore suggested that I could avoid this problem by using both an event hat and a procedure hat, as pictured below:



Calling a Procedure Hat from an Event Hat

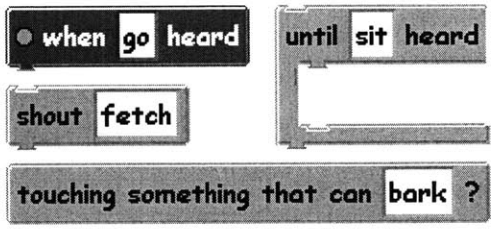
However, it would take a fairly clever programmer to think of this solution, and it is still only of limited help. We have introduced an unnecessary level of indirection by using the auxiliary script. Frequent use of this idiom would make programming in Scratch rather awkward. Ultimately, our problem is only partially solved, because we still need to use a "wave yellow flag" block to trigger

the "fire" behavior. It would be preferable if there were some way to invoke the "fire" script by using the descriptive word "fire" itself.

It seems that what we really want here is (1) to name a useful behavior ("fire") in one object (the bullet), and (2) to use that name to start this behavior from another object (the spaceship). Perhaps we could achieve this by using strings--instead of colors--to name events.

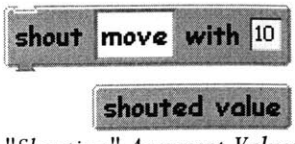
The Shouting Metaphor

We will now explore an event model in which we use strings to name events. The first challenge we faced was the selection of an appropriate metaphor. The obvious choice was to have the various objects "say" event names. However, to emphasize the broadcast nature of events, we eventually went one step further, having the objects "shout" instead. (It is unfortunate that there is a somewhat negative connotation associated with shouting.) Below are some examples of event blocks using this shouting metaphor:



Event Blocks for the Shouting Metaphor

Like the flag metaphor, the shouting metaphor is somewhat awkward when it comes to passing argument values through events:



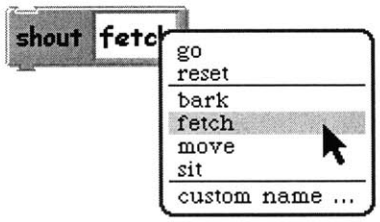
"Shouting" Argument Values

The "Shout Go" Button

We also added the "go" icon (shown previously) on a button in the toolbar at the top of the Scratch screen. Pressing this button broadcasts an event named "go", as if a "shout go" block were executed. This button is equivalent to the one we discussed earlier for waving a green flag.

Selecting a String Name

When our pool of event names consisted of a limited number of colors, selecting an event name simply meant popping up a palette of color choices. Using string names for events, however, means that we are choosing event names from an unlimited space of strings. It is tempting, therefore, to pop up a text field dialog, and let the user enter any string. But, we want be careful not to overwhelm the novice user. Furthermore, we want to prevent the programmer from accidentally introducing errors due to typos. For example, imagine trying to debug a program in which one object is listening for "turn around" but another is shouting "turn arround" instead. I have therefore settled on a pop-up menu (shown below) that allows the programmer to choose a default string name, to select a custom string name, or to create a new custom name:



Selecting a String Name to Shout

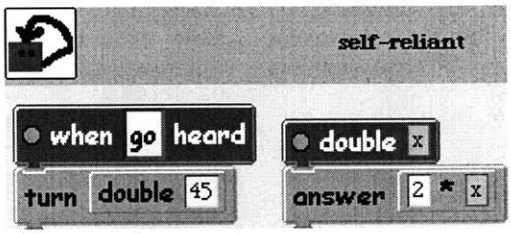
In this example, the user has clicked on the event argument of a "shout" block. This action has popped up a menu showing a number of string name options. The "go" and "reset" names are provided by default. (The Scratch team could probably suggest more appropriate default names.) The "bark", "fetch", "move", and "sit" names are custom names that the programmer has created. These appear in other event blocks in the same project. The "custom name ..." option allows the user to designate a new event name by entering a string into a text field in a pop-up dialog (not shown). Once a custom name is entered in this manner, it will appear in the pop-up menu the

next time the user clicks on an event argument. This mechanism therefore allows a user to type a new custom name on a "shout" block before selecting it later from an event script hat. Alternatively, the user can type in a custom name for an event script hat and later select it on a "shout" block. Additional choices should appear in the pop-up menu for an event script hat, in order to associate a script with a mouse click or key press event. (These choices should also appear for "loop until event" and "wait for event" blocks.)

VIII. SIMULATING PROCEDURE CALLS

Two Ways to Query

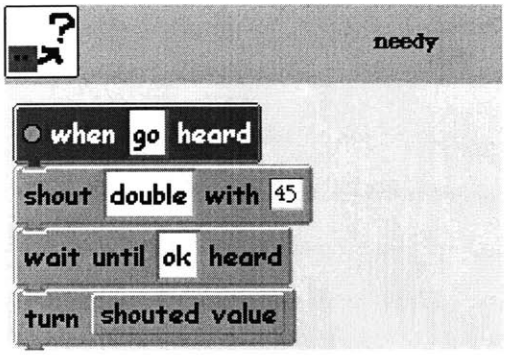
By using strings to name our events, we have now introduced a possible confusion between (1) using events to start scripts with event hats, and (2) using procedure calls to invoke scripts with procedure hats. If the two hats look similar and provide similar functionality, it is therefore reasonable to wonder if we really need two kinds of hats. To resolve this question, we will now take a closer look at these two mechanisms. Consider the following object's scripts:



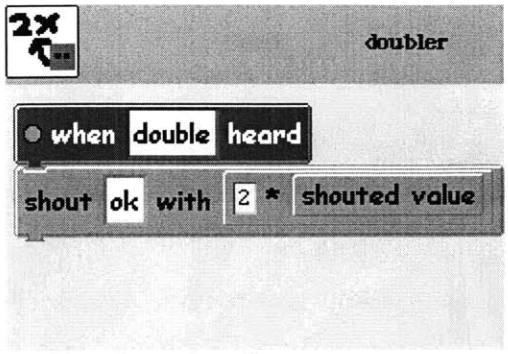
Calling a Helper Procedure

The "self-reliant" object uses a helper procedure named "double" to double the value *45*. The procedure returns *90* to the "go" script, which then rotates the object by 90 degrees. Now, what if this object had not known how to double a number, and needed to query another object to perform this computation? Well, we know that the only way for two objects to communicate is to use events.

The following pictures show an object using events to ask another object to double a number:



Asking Another Object to Double



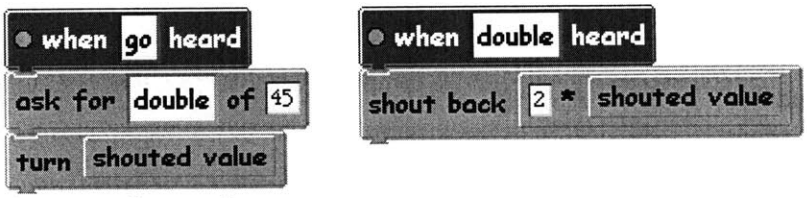
Doubling for Another Object

In this example, the "needy" object broadcasts a "double" event containing *45* as its argument value. The "doubler" object responds by broadcasting an "ok" event with twice this value. The "needy" object waits for the "ok" event, and then turns a number of degrees corresponding to the returned argument value (*90*).

By now, we have seen this trick of using two events to simulate a procedure call multiple times. For the remainder of this document, we will refer to such a simulated procedure call as an *event call*. The cumbersome event call syntax appears radically different from the compact procedure call syntax. Yet, a Scratch user will most likely need to learn both syntaxes, as we expect that using a helper procedure and querying another object for a value will be common operations in Scratch. Notice, however, that an object does not technically need to use the procedure syntax to query itself for a value. It would be perfectly legitimate for the object to use the event call trick to talk to itself. Perhaps a Scratch programmer could get away without ever using the procedure call syntax. It would certainly be desirable if Scratch users only needed to learn one way to query for a value, but using the event syntax is somewhat unwieldy. Can we simplify the syntax?

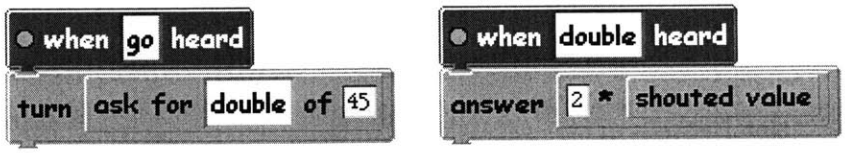
Concise Syntax for Event Calls

To simplify the event call syntax, I decided to conceal the response part of the two-event query. This would at least prevent the user from having to think up two event names. To achieve this, I combined the "shout double with 45" and "wait until ok heard" blocks into a single "ask for double of 45" block. And instead of shouting "ok" to return a value, I introduced a "shout back" block, as shown here:



Anonymous Response Syntax

The "ask" block broadcasts an event, and then waits until it receives an anonymous response event. Under the covers, Scratch automatically generates a name for the response by appending "#response" to the original event name. When the "shout back" block is reached in this example, Scratch broadcasts a "double#response" event with the desired value, and this value is then accessible to the calling script through the "shouted value" block. (If no "shout back" block is encountered, the process will automatically broadcast a "double#response" event with value *0* when the script terminates.) In this syntax, the Scratch user still thinks of the response as a shouted event, but does not need to give the event a name. In the following alternative syntax, the response event is entirely hidden from the user:



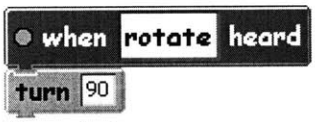
Hidden Response Syntax

Because the "ask" block here returns the value answered by the "double" script, the "go" script does not require a "shouted value" block, and the programmer does not even need to be aware of

the response event. I have therefore replaced the "shout back" label in the "double" script with "answer". Now we have an event call syntax that is nearly as compact as the procedure call syntax, and that can be used to query for values in another object.

Does the additional compactness justify hiding the response event in this manner? By using this syntax, we have lost the metaphor of shouting back an answer. We have also introduced a potential confusion when choosing whether to use an "answer" or "shout" block, which might have been clearer with the "shout back" syntax. Furthermore, it was perhaps more in line with our natural way of speaking to ask for a value before using it, rather than nesting the "ask" block inside the "turn" block. Looking at the hidden event syntax, it may surprise a user to learn that the response is broadcasted to all objects, rather than just sent directly to the process that sent the initial event. (It may therefore be worth considering an alternative implementation in which only the sender receives the response event.)

Sometimes a programmer will invoke a script that does not return a value. For example, we might wish to call the "rotate" script (shown below), and to wait for it to terminate before proceeding:



Rotate Script

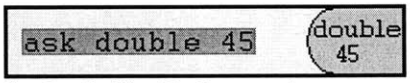
Calling the Rotate Script

For this, we would need a "shout and wait" block, as shown above. Here, the calling process would broadcast a "rotate" event, and then wait for a "rotate#response" event. Scratch would then need to send a "rotate#response" event when the rotate process finishes, even though there is no explicit "answer" block. Now, when Scratch programmers wish to trigger another script, they must decide whether they want the calling process to continue evaluating subsequent blocks immediately (using a "shout" block), or if they want the calling process to wait until the triggered script terminates (using a "shout and wait" block).

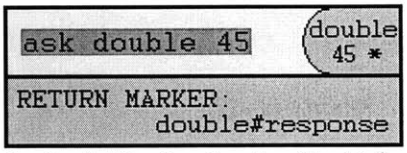
Implementing the Ask Block

Note: Some readers may wish to skip these more technical implementation sections and proceed to reading about more high-level issues in the next chapter. Those readers that do read this section should note that, as before, we will avoid user interface metaphors (such as "shouting") when discussing implementation.

We have discussed a number of primitive blocks in the "broadcast and wait" family. In this section, we will only look at the implementation of a general-purpose "ask" block. (The "shout and wait" primitive mentioned earlier provides a subset of this functionality.) The Scratch evaluator considers the "ask" command (and related primitives) to be a special block. Evaluating an "ask for *double* of *45*" block (for example) proceeds exactly the same as evaluating a "broadcast *double* with *45*" block followed by a "wait until *double#response*". Stage (1) below shows the top of the process stack after evaluating the arguments for the ask block. The process then broadcasts an event named "double" with argument value *45*, and inserts a return marker for an event named "double#response", as shown in stage (2).

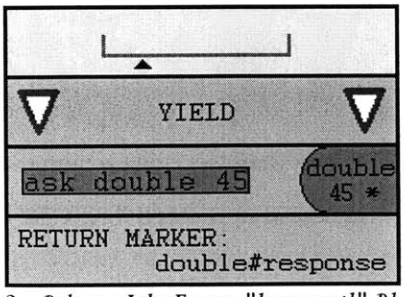


1. Stack After Evaluating Arguments



2. Stack After Broadcasting "double"

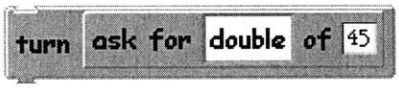
Finally, evaluation of the ask block proceeds as if executing an empty "loop until *double#response*" block, as shown in stage (3):



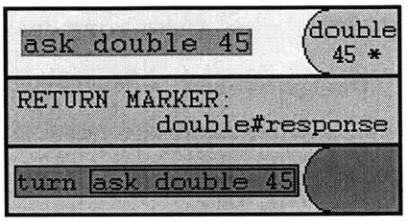
3. *Behaves Like Empty "loop until" Block*

Receiving the Response Event

Suppose the ask block is nested inside another block. Then, when the process finds a response in its event queue, it must pass the response's argument value to the containing block. For example, when a process evaluates the following block, the top of its stack will appear as shown here:

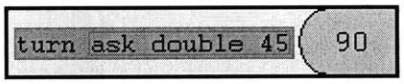


Nested Ask Block



Stack for Evaluating Nested Ask Block

When the "double#response" event is received, the process must pop off all frames down to and including the corresponding return marker. In addition, it must provide the argument value received in the response event (*90*) to the block frame below the return marker:



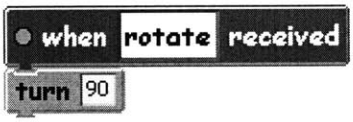
Response Value Passed to Containing Block

Sending the Response Event

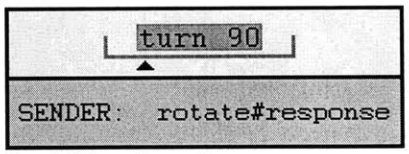
When a script has been triggered by an event, it will terminate and broadcast a response event in one of two situations: (1) if an "answer *value*" block or "done" block (equivalent to an "answer *0*" block) is evaluated, or (2) if the process finishes evaluating the script without encountering an answer or done block. It is simpler to begin by considering this second case. (Note that only process-starting events--those that trigger scripts to start running--will result in response events. Process-continuing events--those that cause a process to break out of a loop--will not result in response events.)

When an event triggers the start of a new process, the process must now store the name of the response event to broadcast. A process started by an event named "rotate" will need to remember to respond with an event named "rotate#response". If no "answer" block is encountered, the process must broadcast an event with the stored response name. This behavior is achieved by placing a special *sender frame* on the process's stack. The sender frame contains the name of the response event to broadcast. Whenever a sender frame appears at the top of its stack, a process will broadcast an event with the given name (and with *0* as the default argument value) and pop off the frame.

Recall that when a process is started, its stack initially contains a sequence frame with the collection of blocks in the script being run. Now, however, when a process is started in response to an event, a sender frame will be inserted below this sequence frame. The following shows the initial stack for a process running the "rotate" script shown:

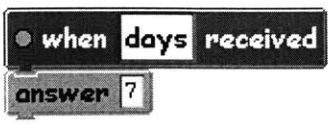


Rotate Script

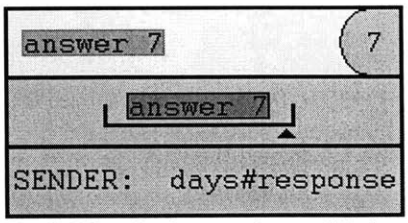


Stack When Script Starts

This inserted sender frame guaranties that the process will eventually send the appropriate response message if no answer block is reached. Now consider the simple "days" script below, in which we do encounter an answer block:



Days Script



Stack When Evaluating Answer Block

When the process reaches the answer block, its stack will appear as pictured above (after evaluating the 7 argument). At this point, the process notes the answer value of 7 (or the default value *0*, if

this had been a "done" block), and pops all frames off the stack until a sender frame is encountered. Scratch then uses the "days#response" name noted in the sender frame, broadcasts an event with this name and the noted answer value, and terminates the process.

IX. REPLACING PROCEDURE CALLS

Having simplified the event call syntax used to simulate procedure calls, it is now time to consider whether we can eliminate procedure hats altogether. We would certainly prefer if Scratch users only needed to learn about one kind of script hat, but what would be the ramifications of removing procedure hats from Scratch?

We should first consider the role that procedure hats currently play in Scratch. Because a procedure and its invocations must reside within the same object, they are of somewhat limited use. In a way, procedure hats resemble an object-oriented language's privately-scoped methods--capturing helpful patterns of computation, but taking a back seat to the public methods (a role played by event hats) that define the object's interface. Used well, a helper procedure can make a program cleaner and more readable. Capturing helpful behavior in a procedure is certainly a powerful idea that we would like Scratch users to learn. But how many Scratch programmers will make use of helper procedures? And of those programmers advanced enough to want such procedures, how many will mind using event calls instead? Because the introduction of events has diminished the role of procedures, maybe Scratch would not be sacrificing too much by giving up procedure hats. To determine the extent of this sacrifice, we will use the remainder of this chapter to explore some of the subtler distinctions between procedure calls and event calls.

Procedure Calls vs. Event Calls

As just mentioned, procedure hats typically serve to designate private helper methods. Consider the spaceship object from our space invaders game. It calls a helper procedure named "hide", which causes the spaceship to hide offscreen. No other object can call the spaceship's "hide" script. If we now change this procedure hat to be an event hat, the spaceship will be able to use event calls to run its "hide" script. But, now any object can broadcast an event to cause the spaceship to hide. Because there are occasions when a programmer wishes to prevent other objects from accessing some portion of an object's behavior, this is a limitation of event calls that we may need to address.

Thankfully, this is a simple defect to fix. One solution would be to allow the programmer to mark an object's event hat as "private". Such a script could then only be triggered by an event broadcasted by this same object. However, it is unclear whether Scratch users would take advantage of such a feature.

A related concern of eliminating procedure calls involves namespaces. Currently, each Scratch object has its own space of procedure hat names. As in our space invaders game, each object can have a helper procedure named "hide", without resulting in any naming conflict. When an object invokes its "hide" script, there is no confusion about which script to run. Now, imagine that we replace each of these "hide" procedure hats with a "hide" event hat. When the spaceship object asks itself to hide by broadcasting a "hide" event, it will now inadvertently cause *all* objects to hide. This problem arises because, unlike for Scratch procedure names, we have only a single namespace for all event names. Ironically, here is a situation where the broadcast model works against our goal of shareability. When importing another programmer's Scratch object into our project, we will need to make sure that its event names will not conflict with ours. (On the other hand, imagine how many more name conflicts we might encounter if we use colors instead of strings to name events.)

There are a few ways to avoid potential name conflicts. One solution is to use a naming convention, in which we include the object's name in any event hat intended for private use. For example, we could use a "hide spaceship" event for the spaceship, a "hide alien" event for the alien, etc. This would be fairly tedious. Our idea of designating some event hats as private could also be used to avoid some name conflicts. Imagine a space invaders game in which each object has a privately scoped event hat named "hide". In such a program, if the spaceship were to broadcast a "hide" event, only the spaceship's "hide" script would start to run. A final and more powerful solution would be to allow a Scratch object to send an event to a single specified target object. Then, an object could send an event only to itself, as a means of invoking a helper script. We will explore this last option in further detail shortly.

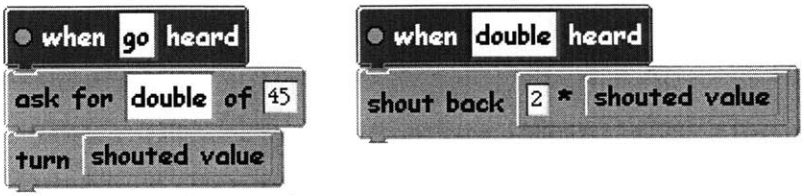
One of the most powerful ideas in programming is the ability to create procedural abstractions that can then be used as if they were primitives. Procedure hats provide this functionality in Scratch. When a script is created using a procedure hat, a new block appears in the object's command palette for invoking this procedure. This block looks and can be used exactly like a primitive block. In this manner, Scratch empowers its users to make their own blocks. If we remove procedure hats from Scratch, we will be depriving Scratch programmers of this ability. Although this issue largely comes down to syntax, it may be perceived as a substantial loss.

A subtle but important difference between procedure and event calls concerns a program's flow of control. In a procedure call, a process associated with one script evaluates blocks in a second script. Meanwhile, the second script may be associated with a process of its own. This confusing behavior was one of the primary reasons for considering alternative communication models like our event system. With event calls, on the other hand, the calling process never leaves the script it started in. Instead, when a process running in one script uses an event call, it broadcasts an event that causes a second process to run the designated script. During this time, the first process is simply waiting for an answer from the second process. When the second process terminates, it broadcasts an event that triggers the first one to resume its work. It is our hope that this clear connection between a process and the script it runs makes the event model more accessible to novice programmers. Even though our event calls are behaving like procedure calls, the user can understand the program's behavior in terms of the event system's more intuitive process model. (This distinction is even clearer when using Scratch's single-stepping mode, which slowly lights up each block as it is evaluated.)

Using Procedure Call Syntax

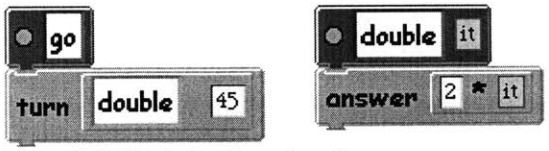
We have now explored several differences between procedure calls and event calls. We might now feel that the gained simplicity of having only one kind of script hat outweighs the advantages of having Scratch procedure hats. If we therefore remove procedure hats altogether from Scratch, an interesting possibility opens up to us. We would no longer need to worry about confusing

event calls with procedure calls. Therefore, we might choose to use the standard procedure call syntax for event calls. In this section, we will discuss the merits of such a choice. Consider again our example event call, using the "shouting" metaphor:



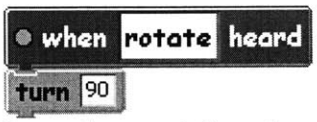
Event Call Using "Shouting" Metaphor

Without procedure hats, there is nothing stopping us from using something resembling procedure call syntax instead. The following blocks show how the same event call might look, using procedure call syntax:

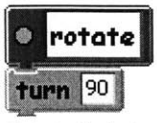


Event Call Cloaked in Procedure Syntax

Behind the scenes, these blocks would still behave the same as those using the "shouting" metaphor. Scratch would still broadcast a "double" event (with *45*) and broadcast back a "double#response" event (with *90*). This is about as concise as we could ever hope to make our event call syntax. Consider our "rotate" script again, shown here with both shouting and procedure call syntax:



Rotate Script with Shout Syntax



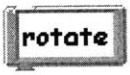
Rotate Script with Procedure Syntax

Recall that the user can choose to invoke this script using a "shout rotate" or "shout rotate and wait" block (shown below), depending on whether the calling process should wait for the "rotate" process to terminate.

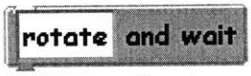
Starting as a Process

Calling as a Procedure

We might be tempted to make the procedure-syntax equivalents of these blocks appear as follows:

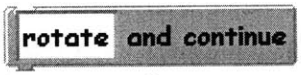


Starting as a Process



Calling as a Procedure

One of the great benefits of this syntax is that the "rotate" block on the left now appears a lot like a built-in Scratch primitive, such as a "turn 90" block. However, a "turn 90" block will cause the object to complete its turn before evaluating the next command. We might therefore be surprised that our "rotate" block does not finish rotating before Scratch runs the next command. (Such behavior would be of even greater concern when invoking scripts that take a considerable amount of time to finish executing.) Therefore, a better procedure syntax might be to label the process-starting block as "rotate and continue", and the procedure-calling block as "rotate":



Starting as a Process

Calling as a Procedure

No More Metaphors?

We have now glimpsed into an intriguing version of Scratch, in which a concise procedure call syntax has replaced our use of more vivid event metaphors. Which of these syntaxes will make programming easier for novices? On the one hand, the compactness of the procedure call syntax allows us to create scripts that are perhaps more readable. There is no metaphor-related text to

distract us from the program's functionality. Also, it is possible that the novice programmer would more easily connect a "rotate" block to the idea of invoking the "rotate" script, than they would associate the wordier "shout rotate" with running the "when rotate heard" script. Perhaps more users will get lost in the intermediate metaphorical medium of shouting or flag-waving than will benefit from the suggested mental imagery of inter-object communication.

No matter how well they are disguised as procedure calls, events will continue to be broadcast to all objects and processes. Users that expect procedure call behavior will quickly be surprised by the quirks of using primitives built on top of our underlying broadcast communication system. Because we cannot completely conceal the broadcast-related behavior (and we do not fully want to do so), it could be beneficial to make the notion of events an explicit part of our syntax. This is something our shouting and flag-waving metaphors did for us. Most likely, these metaphors will help Scratch users understand and internalize Scratch's communication-related concepts much more easily than the compact procedure call syntax. Ultimately, making programming understandable is far more important than keeping Scratch programs short.

Sending an Event to a Specific Object

We might decide that we like the procedure call syntax, but are worried that it is misleading as a user interface to a broadcast-based communication model. If this is the case, then maybe we should consider replacing our underlying event-broadcasting framework with the more traditional object-oriented communication system that this procedure syntax would lead us to expect. By far, the most significant difference between our event calls and a traditional object-oriented system's method calls is that methods are called on a specific object while events are broadcasted to all objects. In fact, one might say that this is essentially the only difference. We should therefore ask ourselves if we ever need to send an event to a specific object.

It turns out that it is possible to invent hypothetical programming exercises that Scratch cannot support without such targeted events. For example, consider the jewel-collecting game we

discussed earlier, in which a human-controlled player moves around the screen collecting various jewel objects, each worth different point values. When the player comes across a jewel, the player needs to know the jewel's value. Now imagine that, in addition to the human-controlled player, there are several computer-controlled characters, each of which is also competing to accumulate valuable jewels. In our earlier implementation of this game, when the jewel collided with the player, it broadcasted a blue event with its value. The player would then receive the event and accumulate the jewel's point value. But now, there are many objects that respond to blue events in the same manner, and we only want the character that just collided with the jewel to respond. Here is a case where we need to send an event to a specific object.

If we were to support such targeted events in Scratch, there are multiple ways this functionality could be added. The simplest would be to restrict an object to sending an event either (1) to all objects, or (2) just to itself. A fancier solution might support a "whisper" block for sending an event to the nearest object that (1) contains a corresponding event hat, and (2) is not the sending object. Perhaps such "local" communication would be sufficient, and there is no need to provide a mechanism to send an event to a specific distant object. (It would certainly solve our jewel-collecting troubles.) The most flexible system would provide an event-sending block that took an argument to identify an arbitrary recipient object. (We would then also require a number of reporter blocks to identify a unique recipient.) All of this would mean a potential explosion in the number of primitive blocks required to support sending events. We wish to avoid overwhelming the programmer with too many options, and therefore a strong argument would be required to justify moving Scratch in such a direction.

Although the hypothetical jewel-collecting game presents a need for such targeted-event functionality, it is reassuring that we have not yet found ourselves needing targeted events in the course of creating a real Scratch project. Therefore, owing to its simplicity and power, broadcasting events to all objects appears for now to be the best direction for Scratch. We will, however, need to keep the possibility of one day supporting targeted events in the back of our minds. If we do eventually require such support, we will need to revisit many of the issues discussed in this chapter.

X. CONCLUSIONS

We have now laid the foundations for a broadcast communication model in Scratch. We have seen how such a model provides the basic functionality needed to support inter-object communication and the management of object processes. We have considered subtle implementation details, and debated the merits of several user interface metaphors. Finally, we have pushed events into the realm of procedure calls, and have raised questions about the role of procedures in our event framework. In this chapter, I will revisit a few of the more important questions raised in this thesis, and present my own opinions as to how those questions should be answered.

1. Can a broadcast paradigm make communication and processes more accessible to novice programmers?

My feeling is that it can, although ultimately this question will be decided by extended user-testing. Broadcasting an event should be one step easier than calling a method in an object-oriented language, because a method call requires the programmer to specify a target object. I believe that this benefit of broadcast communication outweighs the cost of sacrificing some of the power of a traditional object-oriented system. Example projects like the space invaders game indicate that we have not given up the ability to create meaningful projects in Scratch.

2. What basic functionality should a broadcast-based communication system provide for novice programmers?

The heart of such a system should be named events. The core functionality of the system should include the ability to (1) define scripts triggered by events, (2) broadcast an event, and (3) loop (or just wait) until an event. In addition, I believe it is necessary to support events that contain argument values, because these greatly extend the realm of possible projects we can create. The system should therefore let us (1) broadcast an event containing an argument, and (2) access the argument value for a received event. Finally, because simulating a procedure call is a useful construct, such a system should possibly provide advanced primitives for (1) broadcasting an event

(perhaps containing an argument value) and waiting for an anonymous response event, and (2) broadcasting an anonymous response event (perhaps with an argument value) upon terminating a script.

3. Should a broadcast-based programming language for novices include support for traditional procedures?

No, it should not, primarily because we can use "event calls" to simulate procedure calls, and because we can provide clever primitives that imitate concise procedure call syntax. This reduces the role of traditional procedures to their usage as privately scoped helper functions--a small role that could be made redundant by a broadcast system that allowed us to limit the scope of event-triggered scripts. Supporting procedures comes at a subtle but substantial cost. Procedure calls require a tricky process model in which the flow of control for a single process moves between scripts. This model is difficult to represent visually to a novice programmer, and does not appear in a system with purely event-triggered scripts. More importantly, however, supporting scripts that can be called as traditional procedures alongside event-triggered scripts means that novice programmers will need to learn two very different mechanisms that play largely overlapping roles. Every time the user creates a script, they will need to consider which kind of script they should create. It would be preferable to avoid this confusion altogether by omitting traditional procedures from the language.

4. Can an appropriate choice of metaphor make broadcast communication accessible to novice programmers?

My guess is that a metaphor can help a great deal, but only long-term user-testing will tell. Although the language would become fairly compact without a metaphor, it would also appear cryptic to a novice programmer, with procedure call syntax and references to "broadcasting". Our goal is to help the novice enter the world of programming as smoothly as possible, with minimal dependence on teachers and manuals. If a new programmer needs a manual or teacher to suggest they think of broadcasting like "shouting" or "waving flags", then we are better off embedding that

helpful metaphor directly into the language. In addition, these metaphors are ideally suited for activities in which kids act out a program, waving flags or shouting to signal each other's behavior.

5. *Should broadcasted events be named by colors or strings in a programming language for novices?*

I expect that the answer to this question will depend on how long the audience will be engaged with the programming environment. As a two-hour introduction to computer programming, a visual color-based metaphor for events (like waving flags) would be ideal. Such an introduction would involve only simple projects, in which the programmer must only keep track of the meanings of up to three different colors. String names may be too advanced for such an introduction--especially for younger children. As I will discuss shortly, colored events could allow us to provide visual feedback of events as they are broadcasted and received. To provide the same feedback with strings would greatly clutter the screen.

For a user in a sustained programming experience lasting multiple days, I believe it will be better to use string names for events. Ultimately, strings are more readable and meaningful, and will therefore allow a programmer to go much further with the language, by creating more interesting projects with many more event names.

XI. FUTURE WORK

In addition to resolving the important questions raised in this thesis, the Scratch project team should consider several additional event-related tasks. In this last section, we will look at what further efforts could improve upon the Scratch event system.

Additional Event Triggers

One area where further exploration would be beneficial concerns the types of event triggers. In addition to cases where the programmer explicitly broadcasts an event, there are some situations where we might prefer if Scratch broadcasted some helpful events for us. In the space invaders program, we came across one such situation involving mouse events. Initially, we had a script that repeatedly tested if the mouse button was down, and broadcasted an event if it was. Later, we modified Scratch so that it automatically broadcasted a mouse click event whenever the user clicked on the mouse button. We could extend this work to support many other kinds of automatic events, including: clicking on a specific object, pressing keyboard buttons, and detecting when a sensor value has crossed a threshold. Because there are so many actions that could be interpreted as event triggers, part of the work that remains will be to determine which actions should trigger Scratch to send automatic events, and which actions the programmer must repeatedly test for manually (as we originally did for mouse clicks in space invaders).

Making Prototypes and Classes Easier

We can use our event system to simulate an object-oriented language's classes and instances, but doing so is awkward and requires a fairly advanced programmer. The key idea is that the events an object responds to in our broadcast paradigm serve to define a kind of object-oriented interface for accessing that object. Although Scratch does not support classes of objects, we can copy an object to create multiple objects with the same interface. However, if we decide to modify the

implementation of one such object, the others remain unchanged. The solution is to designate one object to be the *prototype*. We only modify the prototype, and make all copies from it. When the prototype is modified, we must delete all copies and make new ones. To achieve this functionality programmatically, we can use Scratch's "copy" and "die" commands. In this scenario, the prototype object copies itself automatically. When we are ready to make a change, we broadcast an event to cause all the copies to die.

Now, the copies have the same interface as the prototype object, so we need to do some extra work to prevent the prototype from dying, too. We therefore add an extra boolean "prototype" field to all objects, which indicates if that object is the prototype or a copy. When we ask all copies to die, all objects--including the prototype--will check if their "prototype" value is *false*, and if so, will die. Now, the copies are completely identical to the prototype, so how can we make sure they have a different "prototype" value than the prototype itself? The trick is that the prototype sets its "prototype" value to *false* immediately before copying itself, and then sets its value back to *true*.

Now, all of this requires a lot of work and a very clever programmer. The payoff of this work is quite large, because classes can be a powerful tool for creating interesting projects. For example, imagine how much better our space invaders project would be if we used classes to add support for many aliens and multiple bullets on the screen simultaneously. It would therefore be worth exploring further in this area, to see if the use of prototypes to simulate classes could be made much easier by having some of the work built into Scratch. Perhaps Scratch could keep track of which objects were prototypes and which were copies, and maybe we could add higher-level primitives to simplify the programmer's work.

Broadcasting to a Network

Perhaps the killer application of the broadcast model would be to allow novice programmers to create networked projects, in which objects from one project broadcast events across a network to

objects in other projects. In fact, broadcast-based communication is commonly used for network communication. The Ethernet protocol is based on this idea, as is Linda (a paradigm in which nodes communicate through a shared "tuple" space) [Carriero & Gelernter, 1989]. Perhaps this is because of the simplicity of broadcast protocols, in which a node can broadcast a message without being certain of which other nodes are connected. For similar reasons, our broadcast paradigm is ideally suited for Scratch network programming.

Imagine that two Scratch users can connect their projects across a network. Then, when one project broadcasts a blue event, the event would be received by any blue-event script in either project. No special network programming primitives would be needed, making networked Scratch programs accessible to the novice programmer.

A fun example of networked Scratch projects would be a network "pong" game. Two Scratch users would run identical (or nearly identical) projects on different machines connected over a network. In each project, a "paddle" object would move horizontally along the bottom of the screen, similar to the spaceship object in our space invaders game. A ball would appear on one of the projects, and the user at that computer would use their paddle to hit the ball up off the screen. The project would then broadcast the ball's state information (coordinates and velocity) to a ball object in the other project. This second ball would align itself with this state information, and begin traveling down its screen until the user at that computer used their paddle to hit the ball back to the first computer. Imagine the power a novice would derive from programming such an advanced project in their first day or week with Scratch!

The implementation to support networked events would be simple. First, the user interface would need to allow a user to connect and disconnect from other projects, perhaps in some peer-to-peer fashion or by registering with a Scratch network server. (There would be some additional work required for a network arrangement in which an event can reach a computer in more than one path across the network.) Then, whenever an event is broadcasted, in addition to being sent to all objects and processes, it will now also need to go to a network manager. This component would manage socket connections, and send event names and argument values as strings to other

connected Scratch sessions across the network. The network manager would also receive events as strings from the network, package them up as Scratch event objects, and broadcast these events to all local objects and processes. One consequence of sending events over the network is that it must be possible to represent all event argument values as strings. In particular, this means that it would not be possible to pass a Scratch object as an event argument over the network.

Adding Visual Feedback

Early user testing of Scratch's event system was conducted using a version of Scratch in which events were represented by flag-waving. Afterward, members of the Scratch project team commented on the difficulty that the kids had using flags. Researcher Margarita Dekoli wrote, "The only thing that was fairly obvious [to the kids] was that in order to make things 'go' they needed to click on the green flag on the toolbar, but I don't think it was because they realized its connection to the green event. Suggestion: We need to indicate in some obvious way which scripts are invoked on different events." This setback suggests that a future version of Scratch using a flag-waving metaphor should provide visual feedback to novice programmers as to which flag is waving and which scripts are being triggered. Research has shown that such feedback can help make programming accessible to novices [Hancock, 2003].

Perhaps Scratch could feature a debug mode (possibly enabled by default) in which a colored flag appears, waves, and disappears in the corner of the screen whenever an event is broadcasted. A colored flash around an object could appear to indicate when it is broadcasting an event. Perhaps, at the cost of slowing down the Scratch program running, such a flash could radiate outward, to indicate the notion of "broadcasting". Whenever an event-triggered script is running, the associated object could glow in the corresponding color. (The display would also need to show when an object is running multiple scripts triggered by different colors.) We could even show what colors an object is listening for, although this may not help as much as it would clutter the screen and potentially confuse users. Finally, it might be helpful if, in some corner of the screen or in an optional extra window, a log of events appeared, with each event scrolling by and shown in

the appropriate color, along with information about who sent the event and what argument value was passed with it. (It should be noted that many of these ideas take advantage of an event's color. It may be difficult or impossible to provide similar feedback for "shouting" or other metaphors in which events are not named by colors.)

Implementing all these features would probably be overwhelming, but a carefully chosen subset could greatly illuminate the behavior of the Scratch event system. Ultimately, continued user-testing can teach us a great deal about what kind of feedback is needed to make broadcast-based programming accessible to novices.

Introducing Events to Novices

If, given the visual feedback described, novices continue to struggle with Scratch's event system, then we may want to explore methods for teaching events to those who have not programmed before--especially for younger children. We should be careful to keep any instruction minimal and engaging, and to encourage students to begin their own programming explorations in Scratch as soon as they are ready. One promising idea is to have children hold actual colored flags (or some object appropriate for whatever metaphor is chosen for Scratch, such as using paper megaphones to "shout" instructions). Children could then act out some simple example programs, in which some kids are taught to behave in specific ways when they see colored flags waved by other children. Such an activity could go a long way toward letting children internalize Scratch's event mechanism. If the activity is done first, then programming may feel like a simple game by the time children are introduced to the Scratch environment. In fact, acting out event-based programs may be an excellent activity to teach programming concepts to children--even without also introducing Scratch.

If, after exploring all these methods to make broadcast-based programming accessible to novices, we achieve only limited success, then we will need to reconsider our choice of communication paradigm. Personally, I believe that any alternative to our broadcast communication model is

likely to be more difficult for novice programmers to learn. As much as we would like to develop an object-based programming language that requires minimal instruction to learn, perhaps we will find, in the end, that there is just no substitute for an encouraging teacher and a formal learning environment, and that there is no shortcut to learning to program.

REFERENCES

[Begel, 1996] Begel, A. (1996). LogoBlocks: A Graphical Programming Language for Interacting with the World. Unpublished Advanced Undergraduate Project, MIT Media Lab.

[Carriero & Gelernter, 1989] N. Carriero, N., and Gelernter, D., "Linda in context," Communications of the ACM, vol. 32, no. 4, April 1989.

[diSessa, 2000] diSessa, A. (2000). Changing Minds: Computers, Learning, and Literacy. MIT Press: Cambridge, MA.

[Guzdial, 2001] Guzdial, M. Squeak: Object-Oriented Design with Multimedia Applications. Upper Saddle River, NJ: Prentice Hall, 2001.

[Hancock, 2003] Hancock, C. (2003). Real-Time Programming and the Big Ideas of Computational Literacy. Unpublished PhD dissertation. MIT Media Laboratory. Cambridge, MA.

[Kay, 1991] Kay, A. (1991). Computers, Networks and Education, Scientific American, September, 1991, pp. 138-48.

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

[Papert, 1993] Papert, S. The Children's Machine: Rethinking School in the Age of the Computer. New York: Basic Books, 1993.

[Resnick, Kafai, & Maeda, 2003] Resnick, M., Kafai, Y., Maeda, J., et al. (2003). A Networked, Media-Rich Programming Environment to Enhance Technological Fluency at After-School Centers in Economically-Disadvantaged Communities. Proposal to the National Science Foundation. MIT Media Laboratory.

[Resnick, Rusk, & Cooke, 1998] Resnick, M., Rusk, N., and Cooke, S. (1998). The Computer Clubhouse: Technological Fluency in the Inner City. In Schon, D., Sanyal, B., and Mitchell, W. (eds.), High Technology and Low-Income Communities, pp. 266-286. MIT Press: Cambridge, MA.

[Steinmetz, 2001] Steinmetz, J. (2001). Computers and Squeak as Environments for Learning. In Squeak: Open Personal Computing and Multimedia, Rose, K. and Guzdial, M. (eds.). Prentice Hall: New York.

CB73-18