The Icicle Programming Environment for Children

Robert Joseph Sheehan

A thesis submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in Computer Science,
The University of Auckland, 2005

ABSTRACT

This thesis describes the design, development and testing of a programming environment for children. Several different approaches were employed to produce guidelines to aid in the design of the environment: a brief examination of the theories of child development, the production of a simple first programming environment, interviews of children to determine what they understood about computer programming and the programs they wanted to produce, and the evaluation of previous programming environments for children.

The resulting guidelines ranged from suggestions as to the age range of children the environment should be aimed at, through to the necessity of making the environment fun.

Icicle (Interaction computing in a constructionist learning environment), the programming environment that was produced, includes several novel approaches to producing and representing programs. Changes in state, or interactions as they are referred to in Icicle, are used as the basis for computation. Instructions are represented by animations, and parallelism is provided in a very simple way.

The system was tested with twelve, nine and ten-year-old children. They found Icicle to be usable and very enjoyable. They produced many programs using Icicle and believed that it enabled them to produce any type of program that they desired. The real limits of Icicle are also discussed.

To Stephanie.

ACKNOWLEDGEMENTS

I wish to thank my supervisors and friends, Dr Alan Creak and Dr Jennifer Lennon, for staying around to see this finished.

I also wish to thank the pupils and staff of Marist Primary School, Mt Albert, for allowing me to come into the school and disrupt things, not just for one study, but for two. The children were delightful and the staff were always friendly and helpful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION AND MOTIVATION

In designing any computer system, it is important to take account of its intended purpose. Of course, the people who design computer systems claim to do this, but in reality design is compromised by a host of practicalities: cost, time, a need for simplicity, compatibility with existing systems and the desire to produce general solutions.

General solutions do not always provide the best match to specific problems. I have presented a course on computer operating systems for many years and one of the interesting questions about each particular operating system is what type of work was this operating system designed to support? Certainly, an operating system controlling the flight system of a passenger aeroplane needs to be different from an operating system for a Personal Digital Assistant (PDA). The aeroplane system must be able to respond within carefully calculated time limits to the changing circumstances of flight, giving absolute priority to tasks that ensure the safety of the passengers and crew. The PDA system must make it simple for users to find and enter information given severe restrictions on input technologies and small output devices.

Some operating systems are designed to support specific or niche activities. Many embedded devices, without the requirements of larger systems, have small and simple operating systems. They do not use virtual memory or file systems or user interfaces. Usually, when we talk about operating systems for use by ordinary people, we mean some sort of general-purpose operating system, such as Microsoft Windows or MacOS or Linux. These systems supposedly provide all the resources we need in order to get our work (or play) done on a computer.

My interest was in computer systems for children. There have been several attempts to produce computer systems and software suites, including the operating system, specifically for schools and other educational establishments. The Plato system (Lyman, 1978) developed in the 1960s was the earliest successful attempt at this. In New Zealand, the Poly system was developed in the early 1980s (The People's Embassy, 1998). One really interesting system, that was only ever implemented as a prototype, was the Pegasus 2000 computer, "A Computer for Children" (Assumpcao, 2000). It included a built-in form of object-oriented Logo and a novel interface inspired by the Pad zoomable display (Perlin & Fox, 1993). Only the Pegasus 2000 system was designed for children. The other systems were designed for institutions – to be used by children, but designed for schools.

MOTIVATION

The question that motivated this thesis was, "What is special about the computing that children do, want to do, or could do, that is not adequately supported by current computing systems?"

With an answer to this question, a comprehensive computing environment for children could be designed. Such an environment would match the requirements of children better than the general-purpose operating systems and user interfaces they currently use. If this environment was not an

operating system, it would at least provide a level between the operating system and the child. All of its facilities would be designed to match the requirements of children: the user-interface, the storage system, the scheduling system, the way programs integrated with the system, connections to the Internet, even the language that the children could use to produce their own programs. The only comparable existing system was the Squeak environment (Ingalls, Kaehler, Maloney, Wallace, & Kay, 1997).

It was clearly too much work to develop a system of that extent. The Squeak environment was the product of many years of development by an extensive team of designers and programmers.

After investigating some of the alternatives, including implementing a prototype of a Web browser for children (Sheehan, 1997), I decided that the programming language component would be the basis for this work.

Of all of the things that children can do with computers, programming is one that cannot be done without a computer. Most other tasks, such as writing a letter, painting, producing movies, listening to music, collecting information, and playing games can be accomplished with other tools.

Being able to program gives a person more control over the functions of the computer. If you can program you can make the computer do the things that you want it to. You are not constrained to the functions provided by others. You are only constrained by the limitations of the machine and your imagination.

CHILDREN USING COMPUTERS

Writing computer software is itself morally neutral; imposing it upon children is quite a different matter. That something can be done, does not mean that it should be. A review of the literature of children using computers and more particularly children programming computers shows very little concern over whether children should be doing this. There is commonly a discussion of why it is advantageous for children to use or program computers, followed by results that show it is possible, and finished by a prediction that this is going to have great effects in the future. Even the great philosophers of computer use by children, such as the MIT school of Papert and others, start from a position of excitement at what they see as the possibilities of providing children with new ways of doing and thinking that can only be done with computers, rather than from a position of scepticism. Cynthia Solomon in her outstanding book on the ways computers can be used in education said, "The computer's presence offers a new opportunity to improve the quality of education" (Solomon, 1986).

Certainly there are papers that "prove" and "disprove" the benefits for children of working with or programming computers (Mayer, Dyck, & Vilberg, 1986; Pea & Kurland, 1984). The current position seems to be that computers can be educationally helpful for children but are more effective when part of a broader educational reform movement (Roschelle, Pea, Hoadley, Gordin, & Means, 2000). Depending on the way they are used, computers can either foster creativity or stifle it (Clements, 1995b). Some computer programs are not beneficial; violent video games appear to encourage violent behaviour (Anderson & Dill, 2000). The evidence is contradictory as to whether

computers are more useful than other technologies, or approaches, at producing learning. This is especially true with early primary-aged children and younger.

WHY CHILDREN SHOULDN'T USE COMPUTERS

Over the last decade, there have been several groups and individuals who have taken the point of view that computers and children should not mix, or at least, not in the ways they currently do. Several books and reports (Armstrong & Casement, 2000; Cordes & Miller, 2000; Healy, 1999; Stoll, 1995) have been published in an attempt to persuade the public that computer use by children should be restricted until more is known about the long term effects of such use.

The natural pace of cognitive development

Critics see the introduction of computers into kindergartens and primary schools as taking time away from other, more natural, forms of development and focusing on the academic and abstract. Many schools (and kindergartens) have reduced recess (play) time, so that children can spend more time working at computers or other academic activities. Critics claim that children should be spending their time with activities that are developmentally appropriate for them.

These criticisms rely on what we have learned about human development. This is not limited to cognitive development, but includes emotional, intellectual, physical, social and moral development. It is argued that the natural process of development is unique to the individual and cannot successfully be hurried. Either something is lost along the way, or the child ends up where they would have anyway, without the unnatural intervention. Therefore, the rush to push children intellectually is wrong, and computer use is seen as doing this. Advocates of Steiner education would agree with this (Setzer & Monke, 2001).

Logic and emotions

Some critics make the simple, but incorrect, connection between the logical, abstract nature of computers, and the uses to which the machines can be put. They believe computer use emphasises analytic, abstract thinking – ignoring the place of feelings in making choices.

I believe that computer use can be creative, stimulating and fun; it depends, at least in part, on the software being used. Even though some may regard word processors as dangerous, they do not restrict the thoughts and ideas transferred to the page any more than a typewriter does.

There are other dangers, related to emotional well-being, that computers pose to some children. As Sherry Turkle says, computers can contribute to escapism and emotional detachment (Turkle, 1995). So can many other things, especially books and movies, and consumerism, and possibly even religion. Admittedly, because of the type of interaction you can have with computers, they are very attractive to people who have a tendency or desire to escape from reality.

Physical dangers

Excessive computer use is detrimental to physical development. Forcing children to sit still for long periods in front of computer screens is bad for several reasons – a child sitting still is not getting

exercise, and is not interacting with real world objects, such as sand, clay, water and wooden blocks. Interactions with such materials are essential for proper physical and cognitive development.

Looking at the computer screen is harmful. It not only forces the eyes to stay at the same focal length for a prolonged period, it severely limits the child's view of the world, presenting everything in a simplified two-dimensional manner. Other physical dangers include the production of noxious gases from computer cases in the first few weeks after manufacture, and the radiation emanating from the screen. Ergonomic dangers, such as repetitive strain injuries, are also a major concern.

Some of the physical dangers associated with computers are certainly worth worrying about. As it is possible for adults to cause themselves permanent damage by prolonged computer use, it is potentially worse for children whose bodies are still developing.

Are these physical dangers any worse than the dangers posed by excessive television watching or even excessive book reading? Sitting still, focusing on the small type in a book for hours on end, can also be seen as a physical danger. The real problem is the cumulative effect of book, television, video game and computer use.

As reported in the "Fool's Gold" report (Cordes & Miller, 2000), Alan Hedge, a professor of ergonomics at Cornell University, recommends that children

> *take a break from computer work every 20 minutes and spend no more than about 45 minutes in any hour at a computer, and avoid spending more than 4 hours a day at computers and video games – including time spent both at home and school.*

Claims that the two-dimensional environment of computer learning is insufficient to produce the stimulus needed for the developing nervous system are also made. A moderate supporter of computer use by children can argue that it is a matter of balance. No one seems to be arguing that children should spend most of their lives in front of screens. Parents, teachers and other adults caring for children should automatically place limits on the use of any particular tool or game. Climbing trees all day could certainly be seen as dangerous. Practising a musical instrument all day exposes children to terrible potential physical harm from repetitive actions.

The stunting of imaginative thinking

Some critics claim that "children in our electronic society are becoming alarmingly deficient in generating their own images and ideas" (Cordes & Miller, 2000). This seems plausible even though there is a lack of concrete evidence. Children who fill their recreational time with professional entertainment in the form of radio, television, video games, computer programs or the Web will only be using a small part of their imaginations; so much is provided for them visually and aurally that there is very little need to fill in the gaps by constructing mental images and even less need to turn those mental images into physical reality or conjure up images of their own.

It is hard to know with any certainty if the children of today are less creative or imaginative than previous generations were. This may be true, but the reasons are many, and removing computers from the children does not seem the sole solution.

Simulating and simplifying the real world

There are problems with programs that simulate reality in order to teach. Simulations employ simplifications and theories that may not coincide with the reality. Apparently, some schools now use software simulations of nature instead of going on field trips to observe the real thing. The reasons given by the schools include: it saves time, it is safer, it is cheaper, and they find it impossible to get enough adults helping to enable them to comply with legal requirements. This problem does not lie with the computers, but with wider aspects of society.

Disagreements over the use of simulations occur at all levels of education. Sherry Turkle describes the battle over the use of simulations at MIT during the 1980s (Turkle, 1995). Simulations of physics experiments, amongst others, were seen as too perfect, without the difficulties encountered when trying the real experiment.

One reasonable principle would be to use the natural world whenever possible and to use the computer only to do things that are otherwise impossible. Rather than run a simulation on growing plants, actually grow them. Rather than always rushing to the Web (and then being distracted by the ease of going off on a tangent), use a book in the library. On the other hand, if you want to study life at the bottom of the sea or conduct a dissection of an elephant, then a computer simulation will be the closest you can get to the real thing. The simulation may not be completely realistic, but information presented in books also includes simplified illustrations, and clear explanations of topics that are always a little messy in the real world.

The cost of computers

The motivation for much of the criticism of computer use by children is the amount of money required to supply schools with computers, software and support. The money spent on computers could more profitably be spent on other things, but it will not be. Fortunately, the computers can still be put to good use.

Many adults like working with computers. Children like working with computers, and the responsible course of action is to ensure that the programs they work with are creative, stimulating and fun, and that they further the children's development rather than stunt it.

THE PROPER PLACE OF COMPUTERS

Whether we like it or not the question, "Should computers be available to children?" is now irrelevant. Computers are used by hundreds of millions of children each day and they will continue to be.

Given that children are going to use computers, can we minimise the harm by giving them something to do with computers that is fun and encourages creativity and problem solving? I suggest that we can, and that one of those somethings is programming. Programming gives children the greatest control over computers. The work described in this thesis is based on the assumption that, given appropriate tools, children can be creative using a computer, producing programs that are interesting and relevant to them.

THE TASK AND THE RESULT

The task was to design and implement a programming environment for children that successfully matched the abilities and intentions of children.

The environment was to be tailored to the programming needs of children. Just as with niche operating systems, unnecessary facilities could be dispensed with and the design focussed on producing a programming environment that produced the types of program that children wanted.

The task was carried out in two parts. There was some background work on children, computers and education, an early prototype, and a formative study. From the results of the work with the prototype and the desire of children to produce games from the formative study, a rule-based system that used interactions as the rule conditions, was settled on. This system was called Icicle (Interaction computing in a constructionist learning environment).

Icicle was produced and tested. The children who worked with it, enjoyed using it and believed that they could use it to make the programs they wanted. From their viewpoint, it successfully matched their abilities and intentions.

The Icicle system makes several contributions to the design of programming by demonstration environments for children, in particular: the removal of the restrictions that grid-based systems have on the placement, orientation and size of programmable objects, the representation of rules with animations, the way in which parallelism is produced and the level it works at, and the method of producing the program rules when new states arise.

C h a p t e r 1

BACKGROUND FOR A CHILDREN'S PROGRAMMING ENVIRONMENT

> *The best software, like the best play materials, should provide a tool that allows children to explore the world creatively, using their imaginations to manipulate and assimilate knowledge about the world around them.*
>
> *from* (Hanna, Risden, Czerwinski, & Alexander, 1999)

Any computer program intended for humans to use can be approached from two directions – from the side of the users and from the side of the task. With a programming environment for children, the task is to produce computer programs; the users are the children.

In this chapter, I examine both the task of programming and the abilities of children. The concept of programmability is introduced and we see how aspects of programmability appear in many different types of programs, including programs that children commonly use. Some techniques used in these programs can be employed in making a more effective programming environment.

Different theories can be used to understand the abilities of children and how a computing environment could be designed to match those abilities. Developmental models are concerned with what children can do at different ages. Theories of learning and user interfaces, including social theories of education, aid in describing the factors that make it easier for children to learn to use a computing environment. The theory of play not only explains why play is essential for children but that fun should be an important part of a computing environment for children.

This chapter includes a brief discussion of theories from each of these categories and a presentation of some guidelines developed from them. The guidelines are not binding, but are supported by experience and a range of arguments. They can be seen as useful starting points for the development of a programming environment for children.

Before looking at the theories and discussing the guidelines, I address what is meant by computer programming.

PROGRAMMING

Most people do not understand what computer programming is and how computer programs are produced. For example, at the beginning of 2004 some source code for Microsoft's Windows operating system was illegally released on the Internet. The commentators and journalists tried to describe what source code was in words that readers and listeners could understand. Terms like "software blueprint" were used to describe the material (Reuters, 2004), demonstrating that the journalists either did not understand what source code is or what blueprints are. A study of senior high-school students who were enrolled in Computing Studies revealed that very few of them,

including some who intended to study Computer Science at university, had any idea of what a programming language was (Greening, 1998).

What is a computer program and how is it produced? The Wikipedia - an online free encyclopedia (sic) with many entries on computers (various, 2001) - describes a computer program by saying, "A computer program tells a computer what to do. Typically, what to do to or with some data." Microsoft's Help and Support Center (sic) in Windows XP defines a program as "A complete, self-contained set of computer instructions that you use to perform a specific task, such as word processing, accounting, or data management." The Oxford English Dictionary definition is, "A series of coded instructions which when fed into a computer will automatically direct its operation in carrying out a specific task."

All of these indicate that programs are associated with controlling the computer. They consist of commands or instructions. The instructions are not given individually by the user, they are somehow grouped together and the computer performs them automatically.

Some programs enable people to write their own programs. These programs can be categorised in various ways. The measure I will focus on is how much control over the computer the user-produced programs provide. What range of programs can be produced? How general or specific are the resulting programs? I will refer to this as the "scale of programmability". All programs can be categorised on the scale of programmability.

Some interactive programs do not allow any programming by the user; examples include simple text editors, painting programs and many games. In each of these, the program manipulates some state (the words in the document, the shapes and colours on the screen, the position of a spaceship) according to actions made by the user (typed or direct-manipulation instructions). In these programs, the state can be a complex object that depends heavily on previous actions of the user. A good example of this is drawing a line. This depends on the existing selection of a colour and a style and the position of starting and end points. Without programmability, users cannot do anything automatically with the program that the designers and developers of the program had not already thought of. If users want to draw arrows, for example they will have to make them by hand by drawing groups of lines. With programmability, the users could produce an arrow drawing routine that makes an arrow when given starting and end points.

At the other end of the scale are programs that are recognisable as traditional programming environments. Examples, from the programming-for-children world, include Logo (Papert, 1980), Squeak (Ingalls et al., 1997) and Boxer (DiSessa & Abelson, 1986). Programs are produced in these environments by the users typing into editors. Most of the code is written and stored in a textual form (even though Boxer uses rectangles to represent encapsulation, the code is still text). The environment can then convert the instructions typed by the user into instructions the computer can execute in order to make some calculation, manipulate some data, request data from the user or some other source, or produce some output on the screen, for example. The amount of control over the computer in these cases is virtually unlimited. Anything that the computer can conceivably do can be produced with such an environment.

On the programmability scale between these extremes are programs that allow the user to demonstrate behaviour; this demonstration causes instructions to be stored by the environment and these can be replayed whenever required. Normally such recorded sequences are called macros. They have traditionally[1] been sequences of key presses, but many environments can record sequences of mouse pointer moves or higher-level actions such as selections and activations of tasks. After spreadsheets, macros are probably the most widely used form of end-user programming. Some macro systems allow the incorporation of conditions in the macros, so that different things are done in different situations, but to use this feature usually requires learning the macro programming language of the environment. Macro programming systems commonly restrict control to the domain of the original program e.g., a word processing program will allow macros that edit text.

End-user programming

Simple macro programming is one form of end-user programming. End-user programming is programming for people who have never taken a programming course and do not consider themselves computer programmers. They are using computers to get their work done or to have fun. Children fit into this category.

A popular approach to end-user programming is programming by example (PBE) or programming by demonstration (PBD) (Cypher, 1993; Lieberman, 2001). The user demonstrates examples of the required action and the environment produces a program from the examples. In many PBD systems there is an element of inference, the system tries to generalize the user's actions so that the program can be used in a number of similar situations. Sometimes the users have to demonstrate the behaviour more than once in a variety of situations. Sometimes they have to accept or reject suggestions from the environment as to whether the generalizations are correct, and sometimes they have to edit a formal description of the program that is really in a traditional programming language.

Since one of the main ideas behind PBD systems is to remove the necessity of learning a traditional programming language many PBD systems do not have a way of displaying their programs to the user; they do not have a visible programming language, but they are still considered programmable.

PBD systems, like most end-user programming systems, are usually constrained to producing programs from a restricted domain, such as drawing, but they do allow the user to have substantially more control over the system than they otherwise would have.

There are other approaches to end-user programmability that are not normally understood as programming that I want to mention because they are commonly used by children.

Game programmability

Many games provide a small amount of end-user programmability. The type of programmability varies, but as many of the games have sold widely, the techniques appear to be easily used.

---

[1] Here I am ignoring the original use of macro-instructions as collections of assembly language instructions.

In many games, there are a series of levels to progress through as you play the game. You usually cannot move to the next level until the current level's objectives have been met. Each level is usually a little more demanding than the previous one, but apart from occasional extra capabilities, the game play on each level is the same. What makes the levels different is the organization of the onscreen objects. For example, the path the player has to traverse could be a maze with a more complicated layout in the harder levels.

Many multi-level games include level editors with them. These are usually the same tools that the producers of the program used, to make the levels in the game. Users can use these editors to construct extra levels for the game rather than being restricted to the levels provided when they bought it. Usually the levels that can be produced have the same game play and differ only in the configuration of objects on the screen. Figure 1 shows the level editor for the Sierra "Lode Runner Online" game. As well as allowing the user to choose the background picture and music to be played during the level, the editor allows the user to place different game objects anywhere on the screen. The objects and their positions constrain, and to a certain extent, define the game play on the level. They usually also define the specific goal that must be reached before the player can move to the next level.

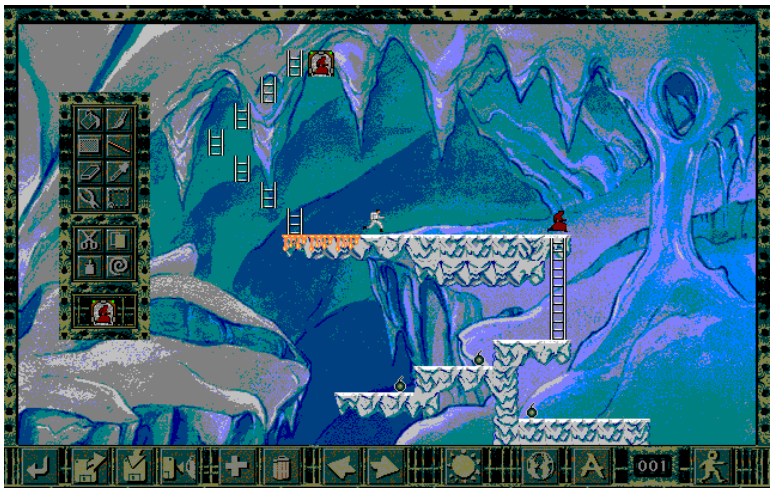


Figure 1 – The "Lode Runner" level editor.

The production of a new game level can be considered as programming. It is a severely limited type of programming; it only works in a very small domain. Only "Lode Runner" levels can be produced by the editor shown in Figure 1. The programming language in this case comprises a vocabulary of screen objects – the ladders, platforms, bombs etc. – and programs are produced from them by placing them on the screen. The mode of interaction switches from designing to playing when the user wants to execute or test the level; just like switching from editing to running a program. When it is played the behaviour of the level depends on where the objects are in relation to each other. This means that the end-user can construct a new Lode Runner game, one that the original designer had not thought of.

This type of end-user programming is also limited in other ways than the domain. The mechanics behind a newly created level are the same as behind all of the other levels. The behaviours of each of

the screen objects were completely determined by the original designers and programmers. The program produced by the end-user is not a program in the traditional sense of a collection of instructions that execute automatically. The levels produced are more like sets of constraints and relations that other rules will be applied to – similar to declarative or constraint-based programming languages. This can also be regarded as a form of programming, if we are satisfied the description of relationships and interactions between objects, define some of the behaviour of the system when the levels are played. In other words, it supplies some of the control that programming gives. In this sense, many other construction programs can also be considered as having a certain amount of programmability.

These games are microworlds. A microworld can be thought of as a simplified model of some physical or abstract system that can be manipulated by a user in order to gain some insight into the system, or sometimes just to have fun. The user constructs something within the microworld and then the rules of the microworld determine its behaviour.

Simulation/construction programs have been around for a very long time. In the early 1980s, Bill Budge produced his "Pinball Construction Set" that enabled users to construct pinball simulations and distribute the completed games to other users.

Hasbro's "Roller Coaster Tycoon" is another example of a simulation/construction game. In "Roller Coaster Tycoon", the user constructs theme parks and roller coaster rides. The game scenarios include targets, such as developing a park of a certain value and maintaining a particular level of customer satisfaction. The construction of a roller coaster (Figure 2) determines factors such as the maximum speed the cars travel at and the g-forces customers experience. From these factors, the program calculates excitement, intensity and nausea ratings. From these ratings, the simulation will send a suitable cohort of simulated passengers to the ride throughout the day, producing various degrees of happiness and satisfaction.

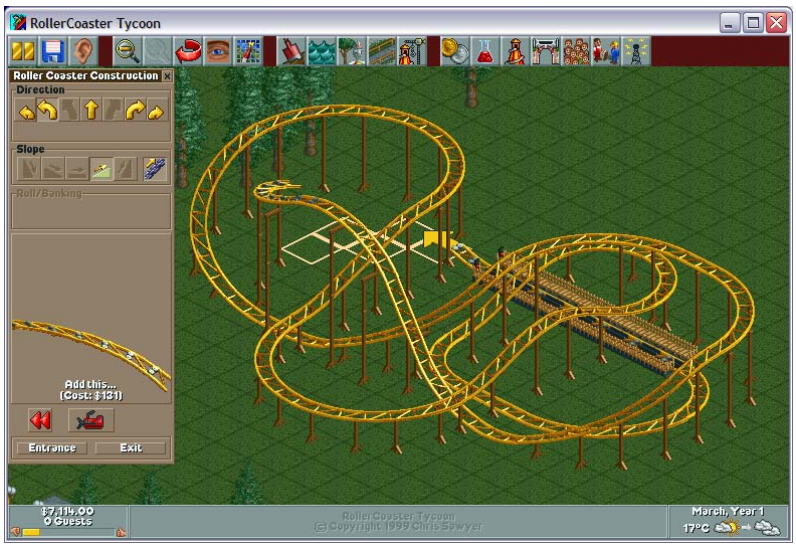


Figure 2 – A roller coaster being constructed.

What makes a "Lode Runner" level-editor or "Roller Coaster Tycoon" game more programmable than a paint program, that allows the stamping of pictures on the screen from a palette of choices, is that the resulting constructions are dynamic; in computer programming terms they can be executed. The thing the user has built, a new level or a roller coaster, can be played or run.

On top of designing and building rides, the user has to decide on a mix of types of rides and other attractions. The overall scores depend on this and other factors so that even though the construction of the roller coaster has a completely deterministic effect on the simulated passengers it only produces a fraction of the total score.

This simulation is played at two different levels. There is the level concerned with running the theme park, setting admission prices and making policy decisions and there is the level involving the construction of roller coasters. Of these, the construction of the roller coasters is most like programming. The language vocabulary in this case consists of the different sections of track that can be used to produce the ride. The behaviour of the cars and passengers on the ride is determined by the design created by the user. This two level approach to simulation is used by many other simulations including the "Sim" series of games from Maxis.

Simulation games such as Maxis' "The Sims" allow the users to program indirectly. By this, I mean that the choices the users make, e.g., to have one Sim (a simulation of a person) talk to another Sim, modify the future behaviour of the simulation. Some of the choices are at the level of providing parameters for the program to operate with and would not normally be understood as programming, but some sequences of behaviour from the user can lead to deep changes in the program results. Actions by the user can teach the Sims particular skills that have an influence in the jobs they perform and the way they relate to other Sims. This ability to control the Sims' lives has led this program to be very addictive for many players.

The "Creatures" series from Mindscape is similar in that it explicitly requires the users to train the onscreen creatures. The creatures learn by being rewarded for behaviour wanted by the user and being punished for unwanted behaviour. Before being able to complete various tasks in the game, the creatures need to be trained properly. This training is a form of programming very similar to that produced by some PBD systems.

Programming games

Other games are specifically for producing programs, but they are sold as puzzle solving or science lab programs. Two classic examples are "The Incredible Machine" series by Sierra and Maxis' "Widget Workshop". "The Incredible Machine" (or "TIM" as it is commonly known) requires the construction of Heath Robinson or Rube Goldberg like machines to perform simple tasks (Figure 3), whereas "Widget Workshop" allows dataflow programs to be produced to perform calculations or carry out amusing tasks. Figure 4 shows a program to calculate the factorial of a number.

These games resemble level editors but are more obviously programming because the behaviour of the screens is determined completely by the programs constructed by the users. With level editors the behaviour is largely determined by the actions of the user playing the level, but as these actions can be seen as input to the level it is more a difference of degree rather than of kind.

Both "TIM" and "Widget Workshop" have parallel programming languages that consist of vocabularies of parts with specific behaviours and ways of connecting the parts together. They are parallel because different parts can be activated and carry out their behaviours simultaneously. Of the two, "Widget Workshop", is more sophisticated, and is correspondingly more difficult to use and was aimed at older children.

Figure 3 – "The Incredible Machine"

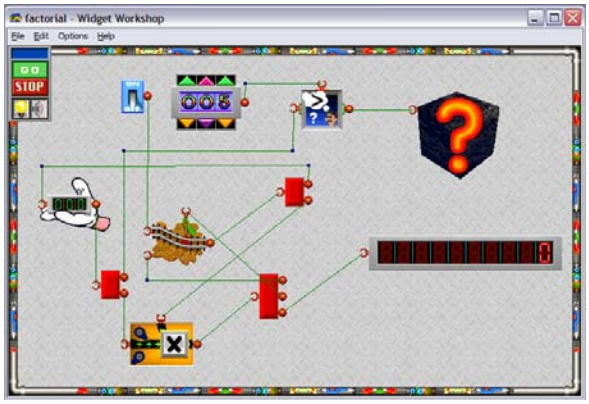


Figure 4 – A factorial program in "Widget Workshop".

Scale of programmability

More programmability means that users have more control over their computers. In particular, it enables them to do things they could not otherwise do, or at least do them faster and more accurately.

Figure 5 shows, roughly, where the different types of program described above, fall on this scale of programmability. There is a large area of overlap between the mid-range categories, due to the variety of programs that the categories cover. The figure is merely intended to show a programmability continuum between those programs that do not allow any programming by the user and traditional programming languages that allow complete control over the capabilities of the computer.

Even if they were not aware of it, children who used games such as those shown in the figure were programming. The programming in most cases was restricted to very specific domains and, as we shall see in Chapter 3, did not cover the range of programs children would like to create if they could.

Because there are many different types of program children would like to produce, we have our first, and obvious, guideline for the development of a programming environment for children.

A programming environment for children should …

> … *be able to produce programs of a wide variety of types, especially the types that children most want to produce.*

Even without knowing the types of program children most want to produce, we see that traditional programming languages such as Logo or Squeak satisfy this requirement because they effectively provide full control over the computer. Whether a PBD environment could be sufficient for the type of programming children would like to do will be discussed later.

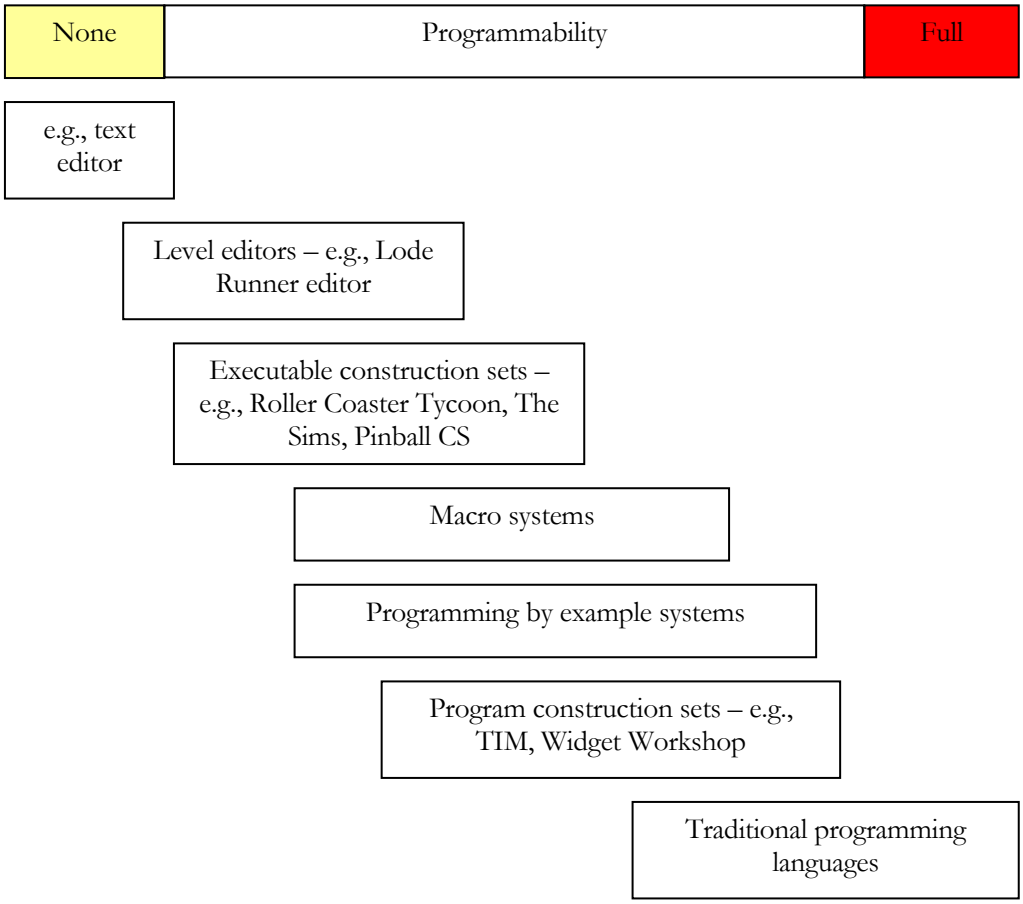


Figure 5 – Programmability of program types.

DEVELOPMENTAL THEORY

No matter how powerful a programming environment is, or where it fits on the programmability scale, its potential can only be fulfilled if it has been designed appropriately for the intended users. When the intended users are children we need to consider their cognitive development.

Piagetian stages

Children develop at different rates in different areas of life: physically, intellectually, and emotionally. Jean Piaget, from the 1920s on, developed the first widely used descriptions of stages of development. Even though his work has been criticised (Egan, 2002) it can still be used as a framework to describe overall abilities at particular ages.

From the point of view of computer programming Piaget's concrete operations stage, typically around the years from seven to eleven (Gage & Berliner, 1984), is particularly interesting. Before this developmental stage, children view the world as a magical place. They believe that movement means life and consciousness. For them "words, dreams and thoughts reside in external objects: the world is filled with forces" (Piaget, 1930).

When a child believes that the computer can do what it wants, because of its own reasons, rather than what it is instructed to do, the child cannot easily see the connection between the commands given to the computer and its behaviour. When the commands are incorrect, the child can reason that

the computer just did not feel like doing what it was told. Actually, many people well beyond this stage, known as the pre-operational stage, still think like this, illustrating that aspects of pre-operational thinking linger on in people at later stages of development.

Because of this animism, coupled with an intuitive rather than a logical approach to problem solving and an inability to view the world from perspectives other than the child's own, the pre-operational stage is not a suitable time for conveying ideas about computer programming to children. There are caveats that must be applied to this statement, and simple aspects of programming have been demonstrated by children before the concrete operations stage (Block, Simpson, & Reid, 1987), but most of the abilities required to grasp the concepts of programming come later.

At the concrete operations stage, children are starting to perform logical operations and are interested in rules – making them and understanding them. These abilities fit nicely with the concepts required when understanding and producing computer programs.

The logical operations that children at this stage can perform deal with concrete objects. "Concrete" in this case means perceptible; they can be seen, heard, touched, smelt or tasted. They are also able to manipulate such objects in their minds and can answer questions about how things would look from different vantage points. This is one example of the general ability at this age, of being able to break out of an egocentric view of the world.

These children are comfortable with cause and effect. Similarly, they can imagine a series of operations and anticipate results without having to perform the operations. This is an essential skill when thinking about the sequence of instructions in a program.

Children at this stage still cannot generalize abstractly but they can use classification schemes. They can play games with complex rules and can apply those rules in given situations. This ability to apply rules to symbols means they can make sense of computer instructions that manipulate data or perceptible objects.

They are interested in how things work and how they are made. This fits nicely with programming construction kits such as "Pinball Construction Set" or "Roller Coaster Tycoon". As they learn best from exploration and manipulation of the environment, a program construction kit that facilitates ease of exploration is appropriate. It should be easy to make changes and to observe the consequences of the changes.

In summary, it appears as though children at this stage of development are ready to learn how to control a computer by programming as long as they are provided with an appropriate environment. From this description, the following guidelines can be produced:

A programming environment for children should …

> … *give the children objects they can see and manipulate to program with.*

The children need to deal with perceptible rather than abstract objects. If there is any state that can be meaningfully changed by the children, that state should be visible.

… *allow the objects to be manipulated in a series of small reversible steps.*

The children understand logical progression and reversibility. By providing the ability to step forwards and backwards through states of the program they should be able to gain a better understanding of the program they are producing.

… *make the sequences of instructions in a program explicit and easy to follow.*

By making program instructions and their sequences visible in a clear and simple way, we capitalise on the ability of the children to think logically with concrete objects.

The Zone of Proximal Development

From a different setting to that of Piaget, Lev Vygotsky proposed that children learn in a social context and he also introduced the concept of the Zone of Proximal Development (Vygotsky, 1978). This zone, commonly referred to as the ZPD, is to do with learning new knowledge or abilities. It encompasses the transition between being unable to perform a task independently, through to independent competence. The idea is that a novice is able to perform tasks beyond his or her current level when collaborating with a more skilled person. It is the interaction with the more able person that enables the learner to acquire new skills or knowledge. Over time, even when the interaction is over, the learner can still perform the new skill as though the collaborator is present.

The lesson to be taken from the ZPD concept when designing programming environments is to provide support for skilled collaborators to work with learners. This can take two forms, the skilled collaborator could be another person, a child or a teacher for example, or it could be the programming environment itself. In the later case, the environment needs to be able to model the understandings of the learner and provide suitable feedback. (This thesis does not deal with the concept and possibilities of such modelling.) In both cases, the environment needs to provide a suitable series of levels through which the learner can move. Certainly, any such environment must have a simple entry point and a gradual learning curve; otherwise, the learner needs too much support from the skilled collaborator.

Gallimore and Tharp (Gallimore & Tharp, 1990) propose six ways to assist the transition through the ZPD:

- Modelling is carried out by demonstrating the skill being learnt.
- Contingency management rewards correct behaviour and discourages incorrect behaviour exhibited by the learner.
- Feeding back lets the learner know how close he or she is to the goal.
- Instructing is giving commands to the learner, with the hope that the instructing voice becomes the learner's self-instructing voice.
- Questioning helps to raise the learner's awareness of his or her current understandings.
- Cognitive structuring provides a model for what is happening.

Contingency management and feedback are provided naturally in a programming environment by allowing the user to run the program. The observed behaviour of the program provides feedback and

also rewards progress, or demonstrates clearly that something is wrong. Modelling can be provided by looking at existing programs, as long as the program instructions are visible and understandable. Instructing, questioning and cognitive structuring are best provided by a skilled collaborator. In Chapter 4 we shall see that Icicle includes a naïve approach to questioning, where questions are used to produce most of the instructions in a program.

Cognitive structure can be supported by the design of a programming environment. Users develop their own models for what is happening in computer programs. There are good models and bad models. Good models help in making predictions and in solving problems, bad models confuse. By careful design and making underlying control mechanisms visible to the user, it should be possible to facilitate the construction of good models.

Guidelines from this discussion:

A programming environment for children should …

> … *be interactive or lively.*

Being able to see immediately the results of changes that they make, helps the children see the connections between their actions and the effects.

> … *provide a variety of ways of using the environment, starting with a very simple entry level and a gentle learning curve.*

Whether a child is working with a skilled collaborator or not, we need to provide incremental steps through the environment leading to greater power and skill. When simpler techniques are understood by the user, only a little effort should be required to advance to a more complicated, general technique.

> … *make its models explicit.*

If the underlying computational model is close to the representation visible in the program, it should be easier for the children to form a good mental model of the system. Every program is an original construction and having a good internal model of the system will help the children to venture into the new construction space.

Play

So far, I have not mentioned the things that motivate children to use computers. In Chapter 3 children report on what they do with computers. The most common category, especially with younger children, is play.

> *Across the world children have entered a passionate and enduring love affair with the computer. What they do with computers is as varied as their activities. The greatest amount of time is devoted to playing games, with the result that names like Nintendo have become household words. They use computers to write, to draw, to communicate, to obtain information. Some use computers as a means to establish social ties, while others use them to isolate themselves.*
>
> *The Children's Machine* (Papert, 1993)

Research into play with computers has accelerated over the last few years. Play is regarded as beneficial to learning (Rieber, Smith, & Noah, 1998) not just as an incentive to do the real work but in the action of play itself. Yasmin Kafai and others (Harel, 1991; Kafai, 1994, 1996) have used the construction of computer games by children to convey important lessons about thinking and teaching.

Play is defined as a voluntary activity with intrinsic motivation. It requires active engagement and has a make-believe quality (Clements, 1995a). It is regarded as essential for the development of children (Goldstein, 1994). It is important for social and cultural reasons; children learn roles and skills they will use as adults, but it also helps children to form models about the physical world. Vygotsky identifies play with the ZPD:

> *Play creates a zone of proximal development of the child. In play a child always behaves beyond his average age, above his daily behaviour; in play it is as though he were a head taller than himself. As in the focus of a magnifying glass, play contains all developmental tendencies in a condensed form and is itself a major source of development.*
>
> *Mind in Society* (Vygotsky, 1978)

Play is more than just developmentally useful, it also appears to have an intrinsic worth to individuals. People engage in play because it is enjoyable. Olaf Damm, who worked for the LEGO company for many years and was regarded as a philosopher of play, had many things to say on the subject.

> *Play is for the most part undisciplined, free, voluntary, and it is fun, and we should never forget that. We can tend to take toys so seriously that we can forget that play must be fun. Otherwise it's not play.*
>
> *quoted in "The World of LEGO Toys"* (Wiencek, 1987)

He also saw the developmentally beneficial aspects of play.

> *Children need to be active. You will never see healthy passive children. A healthy child is always looking for something interesting to do. When such children come to a new place they say, 'What can I use here, what is here for me?' They find and investigate. They want to use their own resources.*
>
> *It is important that the material they get in their hands is not finished, not ready-made. The bricks invite the children to be active, to do something, they call for activity. The various possibilities are endless. Most toys are empty of new possibilities. With LEGO bricks one can always find something new to do. The activity is engaging, fun, absorbing, challenging, rewarding.*

We can then add to our guidelines two to do with play.

A programming environment for children should …

> *… be fun.*
>
> *… not be ready-made.*

This last guideline needs additional comment. For a long time I have believed that children should produce their own constructions, and not merely appropriate the constructions of others. In particular, I have a deep dislike of clip-art included in school projects or web pages. Because of its professional appearance, clip-art discourages children from producing their own artwork. They can never hope to produce work as good as the clip-art. Even worse, many parents and teachers see the use of clip-art as superior as well. It makes things more professional. Children's work should not be professional; it should be the work of children.

Of course, there are places for ready-made constructions. Children who can not physically produce a self-made construction can get enjoyment from ready-made ones. It is also possible to use ready-made constructions as the basis for a different type of construction such as the use of clip-art to produce a collage, or even LEGO blocks to produce a castle.

When it comes to a programming environment, some people would see the use of professionally designed objects as allowing a way in for children who are not comfortable producing amateurish results. Even though I can see this point of view, I believe it is not only developmentally better for children to construct their own objects, it is also more fun.

SUMMARY

By reflecting on different levels of programmability and considering a variety of developmental theories, we have the following guidelines.

A programming environment for children should …

1. *be able to produce programs of a wide variety of types, especially the types that children most want to produce.*
2. *give the children objects they can see and manipulate to program with.*
3. *allow the objects to be manipulated in a series of small reversible steps.*
4. *make the sequences of instructions in a program explicit and easy to follow.*
5. *be interactive or lively.*
6. *provide a variety of ways of using the environment, starting with a very simple entry level and a gentle learning curve.*
7. *make its models explicit.*
8. *be fun.*
9. *not be ready-made.*

In Chapter 2 we will examine the first effort to produce a programming environment that adhered to these guidelines.

LOGO AND THE DESIGN OF ICE[1]

Several programming environments for children already existed at the time of this work. Studies had already been conducted with some of these environments that showed the difficulties children had using them. The environments reviewed included: Logo (Papert, 1980), Boxer (DiSessa & Abelson, 1986), Stagecast Creator - originally called KidSim and then Cocoa (D. C. Smith, Cypher, & Spohrer, 1994), ToonTalk (Kahn, 1996), AgentSheets (Repenning, 1993), NetLogo (Wilensky, 1999), Squeak (Ingalls et al., 1997) and Hands (Pane, Myers, & Miller, 2002).

In order to find out more about programming environments for children at a practical level, I decided to produce a turtle-graphics programming environment designed to solve some of the problems associated with turtle-graphics programming in Logo. This prototype also helped to show the plausibility of some of the guidelines from Chapter 1.

LOGO

Logo towers over all other computing environments for children. It was certainly one of the earliest programming languages designed specifically for children and became the most influential. The reasons for this influence and then almost complete disappearance are interesting and are described in "Towards a Sociology of Educational Computing" (Agalianos, 1996). It was the right thing at the right time, but that time did not last long. Logo is still being used in many schools but it has nowhere near the profile it had in the 1980s.

The creators of Logo were interested in artificial intelligence, and how the writing of programs could be used to gain a better understanding of real world phenomena, including how humans think. In the 1960s, when Logo was developed, Lisp (Winston & Horn, 1989) was the language of choice for many AI researchers. The hope was that if Lisp could be simplified children would be able to use it to construct simulations and other programs.

Early in its development, turtle-graphics (Abelson & DiSessa, 1981; Papert, 1980) were added to Logo in order to provide a mathematical microworld. From the time that Logo was ported to microcomputers in the late 1970s and early 1980s, it was largely known as the turtle-graphics language. Many people seem to associate Logo exclusively with turtle-graphics (Agalianos, 1996) and are unaware that it is a completely general programming language with list processing capabilities.

Problems with the turtle

Children older than 10 have always grasped the concept of the turtle, and how to control it, relatively easily, but for younger children, there are difficulties typing the commands, and more important difficulties with orientation. Fay and Mayer (Fay & Mayer, 1987) found that many children not only confuse left and right, they also believe that turning left means turning towards the left-hand

[1] Sections of this chapter appeared in (Sheehan, 1999) and (Sheehan, 2000)

side of the screen. Younger children did not understand degrees and sometimes thought that move commands meant *move* and then *turn*, or that turn commands meant *turn* and then *move*; for example, LEFT 45 meant turn left and then move 45. Cohen (R. Cohen, 1987) found that the children's understanding of the turn commands was hindered by the instantaneous change of direction of the turtle when it performed a turn.

Control structure problems

Cohen also found young children had difficulty understanding the syntax of the simplest of Logo's control structures, the REPEAT command. One cause of difficulty was the complexity of the command, with a parameter for the number of repetitions followed by a list of commands to repeat. The fact that the number of repetitions was typed before the list of commands, rather than the more common natural language, *repeat THIS 60 times*, also caused problems.

Cohen observed that procedures were not used by young children partly because of the pre-planning required to define the procedure. When encouraged to write procedures the children would command the turtle in interactive mode and copy onto paper the commands they had just typed. Later, these commands would be entered into the procedure editor. Because this observation was widespread, several people implemented Logo programs that automatically recorded the commands typed in interactively and turned them into procedures when children supplied them with names, e.g., (Clements, 1983/84).

In general Cohen and Geva (R. Cohen & Geva, 1989) believed that Logo made too many demands on young children and needed to be redesigned.

> … *the main tenet of this article is that an average six- to nine-year-old child cannot attend to and refine these concepts simultaneously since processing capacity limitations and lack of automaticity may restrict the number of task components the young child may be able to attend to at the same time.*

Technology of the time

At least some of the problems with Logo were associated with the computer technology at the time it was developed. In particular, the only form of input to describe Logo programs was the keyboard. Logo was designed as a simplified version of Lisp. Lisp was popular in the Artificial Intelligence community because of its ability to manipulate programs and its interactivity. Interactivity was seen as essential in a programming language for children. Like Lisp, Logo was a completely text-based language with procedures (functions in Lisp) either typed at a command line or entered into an editor program.

Modern versions of Logo are substantially more complicated, e.g., "MicroWorlds EX" (LCSI, 2003) which allows the production of sophisticated multimedia presentations and other interactive programs without having to type any Logo instructions directly. Logo commands must be typed into "MicroWorlds" only when greater control is required over the turtles and other on-screen objects

than is provided by pointing and clicking. Even so, in "MicroWorlds", the turtle can only be positioned and oriented with the mouse; the mouse cannot be used to program the turtle.

Leogo

One simple improvement to Logo was to allow children to create turtle-graphics procedures by converting mouse drags of the turtle into turn and move commands. Leogo (Cockburn & Bryant, 1997, 1998) provided a variety of paradigms, including direct manipulation to record Logo-like programs in text form. The normal Logo text form and a button and slider form of control were available simultaneously in Leogo. Changes made in one of the forms were automatically shown in the other paradigms. This ability to give commands by dragging the turtle solved the problems of giving incorrect turn commands, and removed the need for young children to randomly experiment with parameters in order to get the turtle to a particular point on the screen.

A BETTER APPROACH TO TURTLE-GRAPHICS

These problems with Logo provided the motivation to produce a turtle-graphics programming environment that not only tried to remove the problems, but also incorporated the guidelines from Chapter 1 in its design. The requirement was to make programming turtle-graphics easier for young children and more fun for older ones.

The Logo problems to be solved were those to do with the commands to turn and position the turtle correctly and easily, the complexity of the REPEAT command and the difficulty of producing procedures.

Design decisions

A number of design decisions were derived from these requirements.

- The system should be directly explorable. Guidelines 2 and 6 supported using simple interaction techniques such as direct manipulation as the first technique to control the turtle. As with Leogo, direct manipulation of the turtle could be used to communicate turn and move commands.
- There should be a progression of control methods. In order for the system to be programmable, the techniques to control the turtle must go beyond direct manipulation. Guideline 6 indicated that a series of techniques that built on each other could be used.
- The dependence on text should be reduced. The system was to be used by young children including those still learning to read. Unlike Leogo, the instructions should not be represented primarily in textual form.
- Concepts of programming should be supported explicitly and automatically when possible. If loops and procedures are produced automatically then the constructs get used, and by making these visible, they can be seen to be useful abstractions – supported by guidelines 4 and 7.
- It should be impossible to produce a syntax error. This was related to the first decision and the removal of text as the control channel, but was also supported by guideline 8 because the

appearance of errors that children do not understand reduces the enjoyment of using the program.

- There should be a comprehensive undo and redo system – supported by guideline 3.

Two of these decisions need to be examined more closely – the progression of control methods, and the explicit support for the concepts of programming.

Incremental steps to programming

If we start with a directly explorable environment and gradually introduce more power and abstraction through a series of incremental steps, as recommended by guideline 6, we can provide an environment with an easy entry point and the power and sophistication of the turtle-graphics component of Logo. The need for explicit instruction will never disappear completely but careful design can make each step towards programming easier. The role of a teacher or peer assisting a new user with the program becomes one of encouragement and facilitation rather than being largely instructional.

As evidence for this idea Cohen designed and tested a series of programs to lead children to the understandings required to successfully program turtle-graphics and concluded:

> … *the acquisition of basic programming concepts by young school children can be facilitated when the programming environments gradually and systematically introduce children to key concepts in a manner that takes into account the child's cognitive and conceptual development.*

Scaife and Taylor (Scaife & Taylor, 1990) also suggested a series of steps helpful in developing the understandings necessary to succeed in the world of textual programming. They insisted on physical computational objects, before moving to iconic, and finally, textual forms of programming.

Concepts required when learning programming

A person learning to program faces many difficulties. Du Boulay lists five problem areas the novice programmer faces (du Boulay, 1989). The first of these is *orientation*, understanding what programming is, and the general approach you have to take when writing a program. (In Chapter 3 a study is described that found out what children understand about computer programming.) Many people find the level of detail necessary to define a simple program daunting, and structuring those details into comprehensible artefacts impossible. These difficulties are made even worse if trying to teach children how to program.

Several concepts need to be understood when starting programming. In particular:

- instructions – a program comprises many instructions or commands. The user needs to understand the level instructions work at. In most programming environments, individual instructions are simple and programs require lots of instructions to do something worthwhile.
- sequences of instructions – normally programs execute one instruction at a time and move through a sequence of instructions.
- delayed execution and stored programs – a program is a sequence of instructions that are carried out automatically some time after they have been entered into the programming

environment. Having the user explicitly command each instruction interactively is not a program. Of course, if those commands are stored and can be executed again then we do have a program.

- making new instructions – defining a sequence of instructions as a new instruction. These sequences are commonly called functions, procedures or macros. Having commands that call sequences of commands is a fundamental abstraction in programming.
- loops of instructions – being able to repeat instructions a certain number of times rather than explicitly having to include the instructions over and over again, is a characteristic of most programs.
- conditionals – being able to make decisions according to the current state of the program prevents programs from merely repeating themselves. Different states lead to different outcomes.

All of these concepts, except conditionals, were incorporated into the different ways the environment could be used. Conditionals were omitted because the focus was on using turtle-graphics to produce pictures and most pictures do not require conditions. In order to be a complete programming environment conditionals are necessary, but as this was an early test of concept the problem of conditionals was left to be dealt with later.

ICE

ICE (Incremental Computing Environment) was the result. The incremental part of the name referred to the different ways the turtle could be controlled. These ways corresponded to different levels of understanding programming.

As has already been mentioned, ICE was similar to Leogo in using direct manipulation of the turtle to develop programs by demonstration, but unlike Leogo it did not record manipulations in text. To reduce the dependence on text, instructions were recorded as a sequence of icons in a history list.

To support the concept of loops, ICE provided a simple inference mechanism that produced loops automatically and represented them graphically. To support the concept of procedures it made the generation of procedures as simple as possible, a single drag of the mouse or a single click of the mouse button.

All commands were given by direct manipulation of the turtle or by clicking on buttons and other controls. This made it impossible to produce a syntax error.

A novice user could start playing immediately with ICE (Figure 6); the turtle could be dragged around the screen, leaving lines behind, and buttons controlling state such as colour could be clicked on. This was the result of the design decision of being directly explorable. The only knowledge required to use the system at this level was how to drag objects and click buttons.

The system could be controlled in a variety of other ways. The progression of control methods was tied in with the concepts required when learning programming. In this way the system could be seen as a progressive series of environments:

- a novel painting environment
- a remote control environment
- a macro programming environment
- a programming by demonstration environment
- and ultimately a general-purpose programming environment.

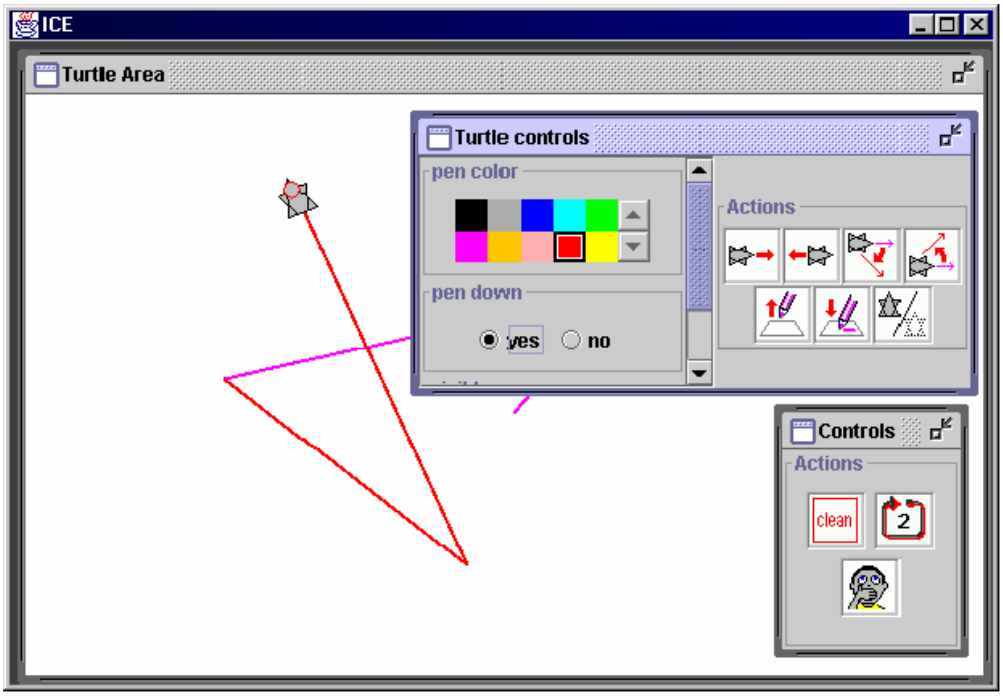


Figure 6 – The first version of ICE.

In the discussion that follows these different views of ICE are described along with the programming concepts they were designed to illustrate.

ICE was never formally evaluated but the comments from children who used it (ranging in age from 8 to 11) were used to modify the design and to lay the ground work for the Icicle system that followed it.

As a novel painting environment – the concepts of instructions and delayed execution

When a child first started using ICE it looked and behaved like a painting application. The difference was that instead of a palette of pens and drawing tools there was a turtle in the middle of the screen. By moving the turtle, trails of paint of different widths and colours could be left behind.

The turtle was directly manipulable. The child could select the turtle and then drag the mouse pointer. During the drag a dashed straight line extended from the turtle to the current position of the mouse pointer and the turtle turned to face the mouse pointer. When the drag action was terminated, by releasing the button, the turtle walked along the dashed line leaving a solid line behind. The dashed

line indicated the path the turtle would follow when the mouse button was released. This made the turtle appear like a straight-line tool. However dragging from the turtle and leaving the line behind were separate actions. Figure 7 shows a line being dragged from the turtle.

Separating the action of dragging the turtle from the result of the turtle moving along the line was important. The turtle moved only after the child released the mouse button, when the cursor was exactly where the turtle should move to. One advantage of this was to show that the total movement of the turtle was controlled by two operations, a turn, carried out when the user was dragging the mouse, and a move, carried out when the mouse button was released; indicating the level instructions work at in this environment. Postponing the turtle's movement was also a step towards conveying the concept of delayed execution, another of the concepts we have to assimilate to become programmers. Interactive text-based programming languages, such as Logo, encourage this understanding because a syntactically correct instruction which has the same form as that required in a program has to be typed before the command is executed in the interactive mode. The command is not executed until the enter key is pressed. In a way, the already entered command is stored, albeit temporarily, in the keyboard buffer. Interactive text-based languages already have this concept of delayed execution built-in. (This advantage of text-based languages is undermined by the requirement for correct syntax, whereas it is impossible to give a syntactically incorrect instruction to the turtle in ICE.)

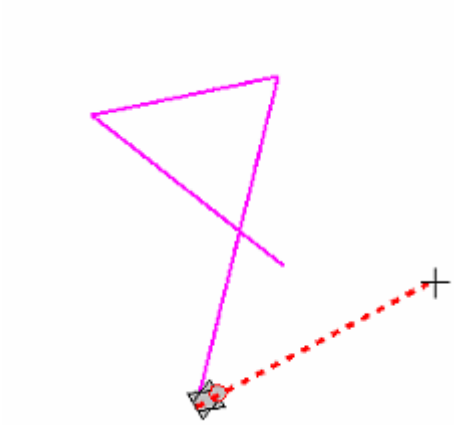


Figure 7 – A drag operation showing the intended path.

In informal testing, children quickly learnt that the turtle responded to commands from them. In practice they had no added difficulty drawing lines with this delayed action. Animating the turtle moving along the line added to the feeling of having given the turtle a command; it was also entertaining (corresponding to guideline 8).

As a remote control environment – the concept of instructions

The concept of instructions was reinforced by treating the turtle as a remote control object that could do one thing at a time. The concept of control without direct manipulation of the turtle was conveyed by clicking on controls in the turtle control panel, shown in Figure 6. The panel included an area with action buttons to remotely control the turtle and an area with properties that could be

changed directly. This panel took the place of the palette of drawing tools in a painting application. Clicking on the action buttons was the second way of sending commands to the turtle. The commands corresponded to the traditional turtle-graphics commands: HIDE or SHOW TURTLE, FORWARD, BACK, RIGHT, LEFT, PEN UP, and PEN DOWN. Having icons on the buttons reduced the dependence on text.

Moving from the direct manipulation form of control to the iconic form reintroduced one of the problems from Logo. Younger children did tend to choose the wrong button to turn left or right if the turtle was facing in the direction opposite to that shown on the buttons. Because there was an undo button this was not quite as bad as the same problem with Logo. Even though it was technically possible to change the image on the icons to reflect the direction of the turtle in the Turtle Area window and reduce the confusion of direction, this was not done because the same icon was used in program listings, and there was no connection between the stored action and the turtle heading at the time of program execution. It was suggested that the icons on the buttons could actually be animations showing the instruction being executed. This partially gets around the problem of the orientation of turns because the change is more obvious; this became important in the Icicle programming environment but was not implemented in ICE.

As an editable macro programming environment – the concepts of sequences of instructions and loops

In the description so far we have what looks like a novel painting program without a lot of the extras that come in such programs designed for children. Children could and did use ICE this way. The next stage was to show the children that their actions constituted a sequence of instructions that could be recorded and manipulated.

ICE maintained a history list of recent actions, shown in Figure 8. This list was designed to show the sequence of instructions in an ICE program. Every manipulation of the turtle was recorded here, including button clicks on the control panel button icons. Drag operations on the turtle were recorded as two actions: a turn to the left or the right of so many degrees, and then a move forward of so many steps. When this history list was shown it provided a third way to control the turtle. The actions were recorded using the same icons as shown in the control panel. Clicking on any action button in the history list re-executed that action, adding a new copy of it to the end of the list. In this way a child could drag the turtle a particular distance, and then get the turtle to move exactly that distance again by clicking on the button representing the move in the history list. This made it easy to draw perfect squares even when the first side had been dragged by direct manipulation.

Most forms of programming require a formal description of a sequence of events. These formal descriptions are usually perceptually distant from the events they represent. This has been referred to as the *gulf of representation* (Wright & Cockburn, 2000). In the Cognitive Dimensions Framework (Green & Petre, 1996) this distance is a value of the *closeness of mapping* dimension. Even in ICE there was a gap between the actions of dragging the turtle and the representation of the drag as two iconic buttons in the history list. However this distance was minimised because the buttons were the same in

looks and behaviour as those in the control panel, and the child had been using those buttons frequently. Closeness of mapping became very important in the follow-up Icicle system.

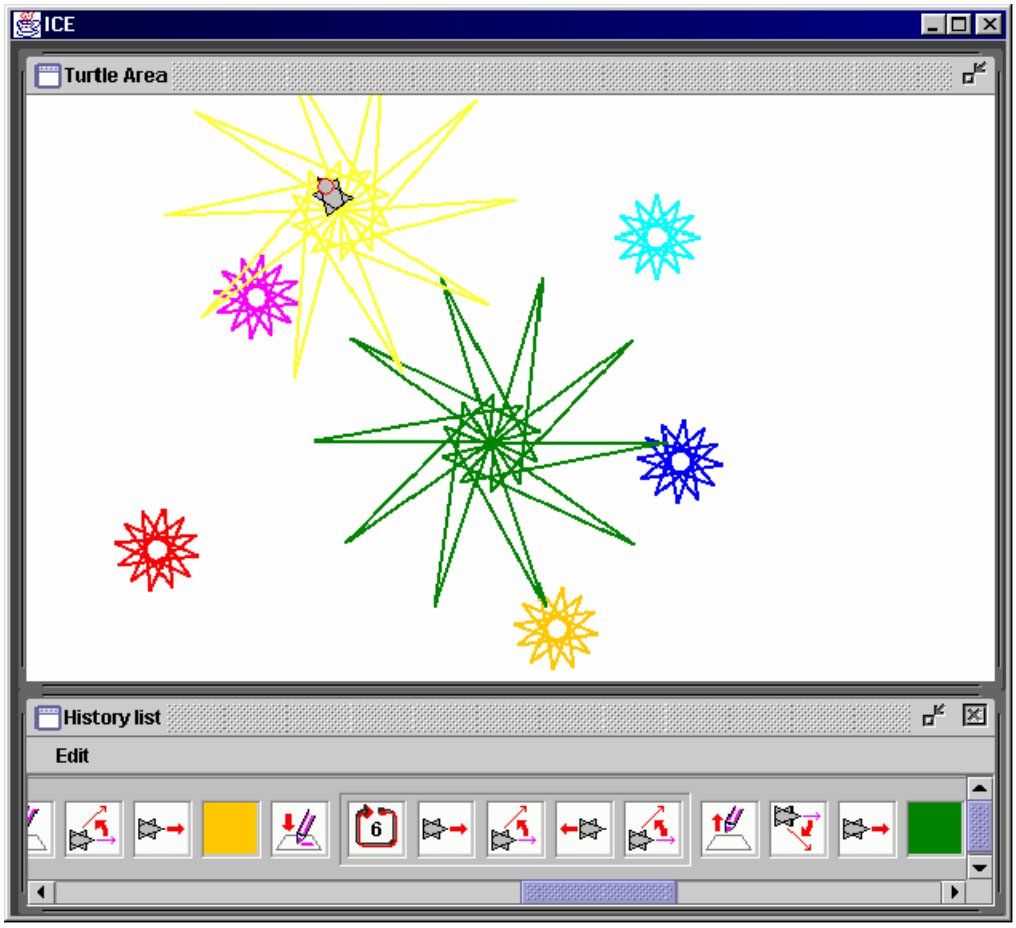


Figure 8 – The history list of commands at the bottom of the ICE window.

The history list was not just a list of previously executed actions; it also provided the basis of the automatic loop recognizer. If a child repeated the same sequence of actions - e.g., moving forward followed by a right turn followed by moving forward followed by a right turn - a loop recogniser spotted this and represented the actions in a loop box. The loop box was the equivalent of the Logo REPEAT command. It was an attempt to avoid the problems with the REPEAT command and to demonstrate the powerful concept and use of loops.

A loop box was a recessed box surrounding one copy of the list of repeated actions, preceded by a loop icon button, inside of which was a count showing how many times the actions had been repeated. See Figure 8 for an example of a loop box (it starts with the icon containing the number 6). Each time the actions were repeated the loop count increased. The loop icon button could itself be clicked on and the entire loop box was executed again. If the loop count was 6 then clicking on the loop icon button meant the instructions after the loop icon were once again executed 6 times.

This simple process could be used to construct complex drawings with a very small number of user actions. The loop recogniser automatically collapsed repetitions into loops and worked recursively, i.e., if a loop was created that matched a previous loop a new outer loop box was generated with a loop count of 2 containing the original matched loop.

It was very common for children to keep clicking on loop buttons because of the large visual changes this produced for little effort. The easiest way to get this effect was to draw a few lines, possibly changing colours along the way. Then, click on all of the actions in the history list one after another until the loop box was produced. Then, return to the front of the loop box and click on the loop icon button. (It would have been better if the loop icon button was at the end of the loop box, but this did not occur to me at the time.) This caused the entire loop to be carried out again, giving rise to a copy of the loop. The loop recogniser turned these two adjacent loop boxes into the contents of a higher order loop box. Continuing in this way, children frequently produced loops repeating the original instructions a power of two times.

In terms of motivation, this was similar to the effect common in Logo programming where children type FORWARD 10000 in order to see the whole screen covered with lines. I refer to this phenomenon as *large effect for little effort*. It is apparent in lots of software for children. I believe this helps explain the popularity of drawing programs such as *KidPix* (Broderbund, 1991-2003) where many of the tools create this type of effect.

The loop recogniser was designed to help develop an understanding of repetition and to be simpler than the Logo REPEAT command, but proved to be useful as a tool simply for its entertainment value.

Figure 9 – The improved ICE interface, the *squiggly* mode button looks like the letter S.

An improved version of ICE

While developing and testing this section of the program the user interface was redesigned to make it more colourful and to provide larger buttons so that younger children could click on them more reliably (Crook, 1992). Figure 9 shows the improved interface. Other aspects were changed after suggestions from children users, in particular to do with turtle manipulation. These improvements will be discussed at the end of this section.

As a programming by demonstration environment – the concepts of stored programs and procedures

With the ability to produce a whole sequence of actions by clicking on one button, the system was introducing the concept of stored programs to the child. The automatic repetition of commands caused by clicking on the loop icon button, demonstrated that not only had the previous commands been remembered, but that they could be played-out again at a later time.

Actions in the history list could also be edited. If an action was altered or deleted, the drawing was automatically altered to match the current history of instructions. It was just as if the new list of instructions was the sequence originally performed by the user. The child could easily make corrections to the existing drawing; maybe the tree would look better with purple leaves. This introduced the concept of debugging but did have an unfortunate side-effect; a small change to a value early in the history list could lead to dramatic changes in the final drawing. Although this was difficult for children to understand they still enjoyed playing with it because of the large effect for little effort.

At this level the system could be thought of as allowing a child to record one (possibly large) sequence of instructions with a loop recogniser to simplify and add structure to the history list.

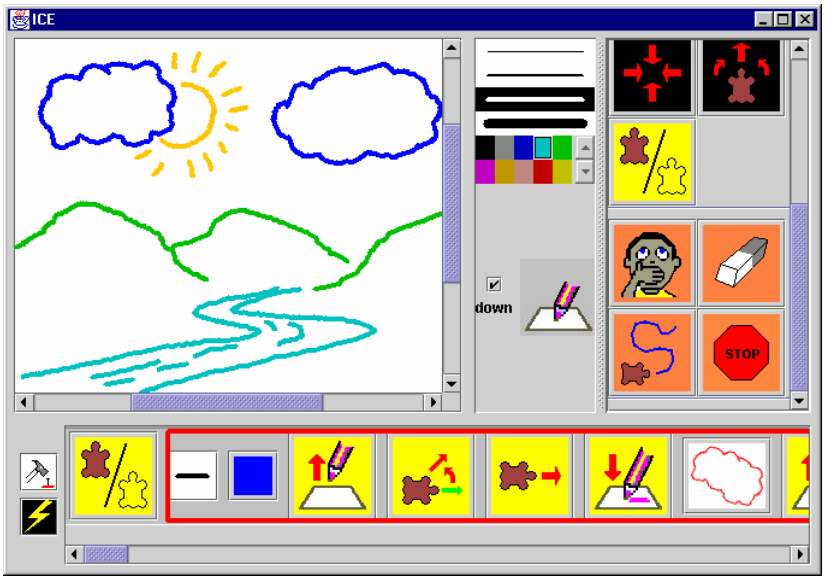


Figure 10 – The red line indicates the selection of a sequence of instructions.

The next level of the design dealt with the concept of procedures. Once again the history list was the basis for illustrating the concept. A group of contiguous instructions could be extracted from within the history list and turned into a single action or procedure. By dragging the mouse over a contiguous group of action buttons from the history list, they were stored in their own window, and the child could re-execute the sequence with a single click. The list could also be edited. The sequence was *named* with a reduced icon of the drawing it produced. This new command was then added to the turtle's control panel and could be invoked in the same way as a built-in command. Figure 10 shows a sequence of action buttons being selected in the history list, the red box indicates the actions being

chosen. When the actions were selected, not only did the new command get added to the control panel, but it also replaced the original sequence of action buttons that were in the history list as shown in Figure 11. In this example, the new action drew a complete scene of mountains, river and sky. Clicking on the corresponding action button would draw the scene again starting at the current position and orientation of the turtle. The children who worked with this version of ICE found the iconic representations of the new actions easy to understand and use. As could be expected, repeating complex scenes was another commonly performed action with large effect for little effort. Figure 12 shows the effect of nine repetitions of the landscape action from Figure 11. It also shows the window listing the instructions of that action.

These collections of actions were effectively macros or simple procedures. In some ways this was similar to the Chimera system, a graphics program that used an editable graphical history list to produce macros (Kurlander & Feiner, 1992). The histories in Chimera represent the state of the display at different time periods, initially after each interaction, but they can be compressed in a way vaguely similar to the method ICE uses to represent a number of commands by the resulting picture. ICE is different however, in the way it shows actions, not state, in its history list. ICE histories are not "storyboards" from animation, film or multimedia design. Chimera was not aimed at children and did not use turtle-graphics. Because of this it was easier in ICE than in Chimera, to produce simple stamp like macros that could be positioned anywhere, in any orientation,

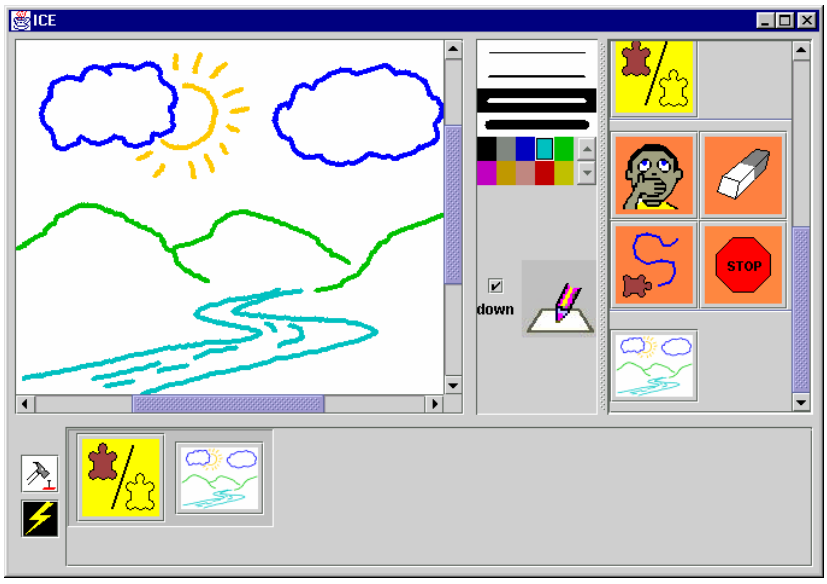


Figure 11 – The selected commands now appear as one instruction in the history list and the control panel.

Even though the relative positioning was helpful in this way it made it difficult to do other things. As an example, if a child wanted to draw a house with four windows and had made a procedure to draw a window, then positioning the windows in the correct places, required the turtle to be moved to the place to start drawing the window. This was difficult because the child had to remember where the turtle was relative to the completed window when the procedure was originally produced. Was it at the bottom-left corner of the window or was it in the centre, etc? The turtle also needed to be orientated correctly, otherwise the window would be drawn at the wrong angle.

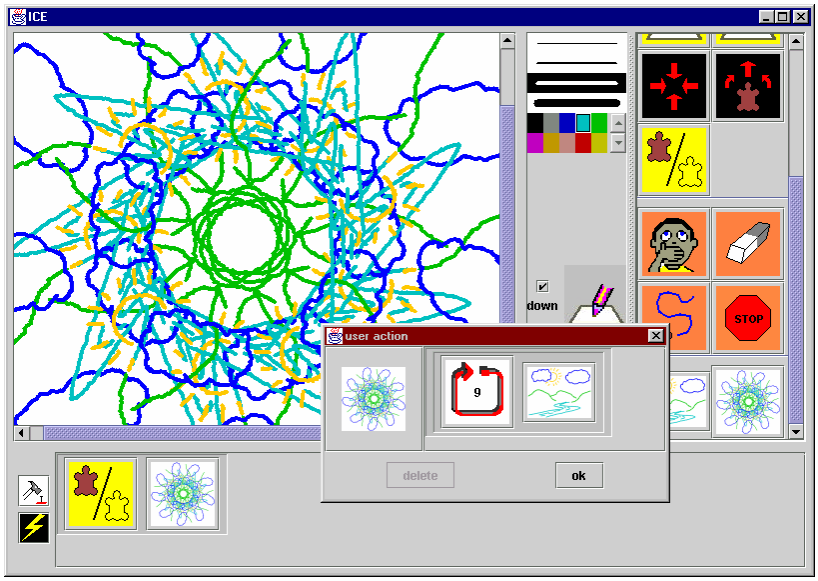


Figure 12 – The result of repeating the procedure nine times.

Since procedures could be constructed entirely by turning and dragging the turtle, at this level ICE could be seen as a simple programming by demonstration system. It was different from a traditional programming by demonstration system because the list of commands was shown as it was developed and could be edited at any stage. Some programming by demonstration systems can show the user the source code generated by the user's actions and some do require editing, but usually there is a perceptual gap between the actions and the source code. In ICE the same icons used to convey the original commands were used to represent the source code, thus reducing the perceived gap between user actions and the recorded programs. Unfortunately, the standard move and turn icons did not convey information about the distance to move or the angle to turn through, except with the use of tooltips. This needed to be more explicit.

The only inference that was made by the system was the recognition of loops. This was simple for the children to understand.

As a general purpose programming environment

Originally ICE was intended to become a general-purpose programming environment. However, the design to add conditionals to ICE was cumbersome and the one thing children who used ICE wanted to do most of all was to create interactive programs, especially games. So the lessons learnt from the design and implementation of ICE were combined with the results of the study I conducted on children's understanding of computer programming and eventually came to fruition in the Icicle programming environment.

Improvements to ICE

Children found it frustrating when first coming to ICE to do simple tasks that required unconnected lines. Think through the operations required to produce the lines in Figure 13. After the first line has been drawn, the pen must be raised before the mouse is moved to the start of the new line and then it must be lowered before the second line is drawn. In the original version of ICE, the

PEN UP and PEN DOWN commands must be explicitly selected by the user at these points. Many children would move the turtle between the lines without first changing the state of the pen. Lines were drawn when they were not intended, and vice versa if the user had moved the turtle with the pen up and now wanted to draw. As these were common actions (that were not envisioned earlier) and common actions should be easy to perform, a solution had to be found for this problem.

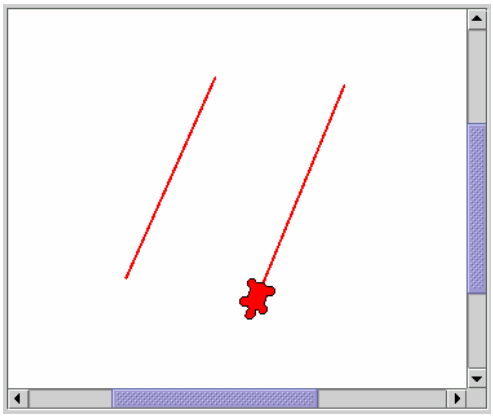


Figure 13 – What actions are required to produce these two lines?

In the original version, pressing the mouse button when the pointer was not over the turtle did not do anything. In the revised version, pressing the mouse button in this situation caused the turtle to do a PEN UP followed by a TURN and MOVE to the position of the pointer and then a PEN DOWN. These instructions appeared in the history list, showing the user exactly how the system was representing the action. This simple addition was a great improvement.

The children also requested a totally different approach to drawing complex shapes, because they found it difficult to make the turtle produce curves, not just straight lines. This was only possible by approximation if the child was very patient and had very good control of the mouse. By repeatedly clicking, dragging and releasing, a large sequence of small lines could be made to look like a curve.

To facilitate this type of curve a new tool was introduced to change the state of the mouse. The mouse was either in *squiggly* mode, and the turtle would draw lines exactly as the child moved the mouse, or in *straight* mode where the turtle tracked the user's movements, but did not record the turn and move until the user had released the mouse button. The *straight* mode was the way mouse input originally worked. In the *squiggly* mode, the user's motions were recorded in the history list with an icon showing the curve as produced on the main turtle display rather than by a very long list of turns and moves. Figure 9 shows a drawing produced with the *squiggly* mode. These actions could then be repeated exactly by clicking on the icons in the history list.

Generally, different modes of interaction are not regarded as good user interface design, however it was very easy to get used to the *squiggly* and *straight* modes. The *squiggly* mode was very popular with the children users.

Using ICE

It was possible to produce very complex shapes in ICE – shapes that required a great many instructions in a traditional programming language. Children could concentrate on combining high-level shapes in their programs without losing the advantage of fine control over turtle movements when required.

A favourite project was to write letter procedures and then use these to write words with the turtle (Figure 14). This was not as trivial as it appears because many uppercase letters were represented as sequences of curves that had to be combined to form a correct procedure. It was of course much simpler than doing the same thing with Logo. This raised the interesting question of whether the floor was too low. By making things easy for children, they could go further before having to think or plan their programs. Thinking and planning were still necessary at some stage, as the tasks the children attempted became more complex. The ease of experimenting in ICE made it more appropriate to those children who favoured a soft (Turkle & Papert, 1991) approach to programming and learning, while still catering for the hard approach. In fact, the *squiggly* and *straight* modes were an apt metaphor for this. I will return to this idea in Chapter 4 when I discuss bricolage (page 75).

Figure 14 – Using ICE to write.

LESSONS FROM ICE

ICE demonstrated solutions to some of the problems associated with traditional turtle-graphics in Logo. The automatic production of loops and the simple production of procedures was successful in so far as they were used freely and frequently by children. The guidelines from Chapter 1 were useful.

ICE was certainly easy for children to use; though whether the design encouraged a greater understanding of the concepts of computer programming was not tested formally. The children who used it certainly made connections between the actions they performed, either dragging the mouse or clicking on buttons, and the buttons that appeared in the history list.

ICE was also useful in that it led to a better understanding of what a programming environment for children should provide and various other useful observations. ICE had some characteristics that were positive and some that were negative, but they were all helpful in the following work.

Positive lessons

The low floor approach. Children found it easy to get going in ICE. In its revised version it started off working very much like a simple drawing program. Any programming environment for children must make it easy to get started. There must be very few things to learn before being able to produce something worthwhile in the environment. This was a reinforcement of guideline 6.

Display source code as it is being produced. Having the source code (the actions in the history list) appear as the child produces a program makes the connection between the actions and the listing clear. This matched guidelines 3 and 4.

Relative positioning of objects is more powerful than absolute. Even though there were some problems with the accurate placement of objects drawn by user defined procedures, the ability to draw objects relative to each other made it easier to aggregate procedures into higher level procedures that could then be executed in any position and with any orientation.

Negative lessons

Source code should be readily understandable even to a novice user. Some of the icons in the history list were immediately understandable to all users, e.g., a coloured square to represent a change of the pen colour. Others were partially understandable, e.g., with the turn icons it was clear to most children that they were turns but whether to the left or right was not clear. Even worse, the angles in the turn operation were not understood by many of the children – matching the findings from Logo. In order to find out how far a turn or a move was, the child had to move the pointer over the history list and wait for a tooltip to appear.

Make interaction possible. ICE was severely lacking in the ability to create interactive programs, like games. It could be used to provide animation by drawing a picture, erasing it and drawing a different picture, but in the implemented versions of ICE there was no way for a person to control the turtle when it was running an ICE procedure. There was also no way for the turtle to make decisions, e.g., to do something different if it was over a blue section of the screen. Even though children enjoyed playing with ICE, it did not do all that they wanted to be able to do, violating guideline 1. I needed to find out more about what children wanted to do with a programming environment.

WHAT DO CHILDREN THINK COMPUTER PROGRAMMING IS?[1]

In order to gather information that would assist in designing a better programming environment for children, I tried to find out what children already understood about computer programming, and the response they had, both to computer programming and to people who program computers. By building on the understandings children bring to the task of computer programming, I hoped to design an environment that makes the experience of programming more rewarding for children, and easier for them to use, to produce the types of program they want. I also wanted to find out what types of program those are.

The main research questions were:

- What attitudes do boys and girls have towards computer programming and computer programmers?
- What do boys and girls understand about computer programming?

An understanding of these attitudes could be used in the design of a programming environment to reduce negative feelings towards programming and widen the appeal of computer programming to more children.

Just as many stereotypes of science and scientists are portrayed in the print and broadcast media, so are stereotypes of computer programmers. I wanted to find out what perceptions six and ten-year-old children have of these topics. Do they see computer programmers as white-coated technicians concerned with the technology, rather than with the people who use the technology? Or do they see them as artistic, creative people using a new technology to represent ideas and provide entertainment in ways not possible without the computer?

By the nature of the questions and the techniques used to extract the information I was restricted to making this a qualitative study with a small number of students, and hence statistical tests were not appropriate.

Many studies have focused on children's attitudes towards computers and computer users (Barba, 1990; Brosnan, 1999; Levin & Barry, 1997), but not specifically on the children's understandings of computer programming and computer programs, nor on the perceptions the children had of the people who produce computer programs. The expectation was that young children would have very little understanding as to how computer programs are produced, that the understanding would improve with older children and that the children would hold stereotypical views of computer programmers.

[1] Much of this chapter appeared in (Sheehan, 2003a).

BACKGROUND

This study was inspired by the work on children's images of science by Aaron Brandes (Brandes, 1996) who investigated what children believe science to be and the image they have of scientists. Being a technological activity, computing should share in some of the perceptions children have of science, but because computers are so common in homes and schools it seems reasonable to expect differences as well. Certainly, the concept of computer programming and what computer programmers do, and are like, was not touched upon by the Brandes study. The original study used two main instruments: children's drawings of scientists at work, and individual interviews. A similar approach was used in this study.

The study of drawings to discover understandings has been used for many years, at least as far back as Piaget's work in the 1920s (Piaget, 1930). As reported by Barba, McMeniman was the first to use children's art to assess attitudes (Barba, 1990). There have been many "draw a computer user" studies – several to determine attitude differences between girls and boys. Brosnan reports that significantly more girls draw males using computers, than boys draw females (Brosnan, 1999). Drawings were also used in an earlier study to capture children's understanding of computers and how computers work (Denham, 1993).

THE STUDY

The school used for the study was Marist Primary School, Mt Albert, an integrated Catholic school within the New Zealand state education system. The pupils come from very diverse backgrounds, many of European and Pacific Island descent but with a sizeable proportion of recent immigrants to New Zealand, mainly from the Indian sub-continent region.

Two classes were selected to take part in the study: a year 2/3 composite class (six and seven-year-olds), and a year 5/6 composite class (nine and ten-year-olds).

A general discussion about computers was conducted in each class before the children were asked to draw a picture of a person programming a computer. The discussion was to focus the children's attention on computers, and the things computers are used for by people. A week later, individual interviews were conducted with a selection of the children to ask about computers and programming. The children were also asked to comment on the pictures they had produced in the first part of the study. Thirty-six children took part in the first part of the study, drawing pictures, and twenty-seven children were interviewed individually for the second part.

Classroom discussion

Twenty-one children were involved in this part of the study from the class of six and seven-year-olds. Fifteen children were involved from the class of nine and ten-year-olds. The classes were chosen because they comprised a good cross-section of the children in those age categories. The small class size in the older age group was the result of fewer than anticipated approvals for participation.

The first session with each class began with the discussion to get the children warmed up and thinking about computers, before they were asked to draw someone programming a computer. The

classroom discussions were audio taped. The opening question was, "What is a computer?" It was obvious that to these children, computers are almost as common as telephones or televisions and they have always been around. For example in one of the follow-up interviews, when a 6-year-old girl was asked where computer programs come from, she said, "Your Mum might have one from when she was little." In the interview section, we shall see exactly how common computers were in the lives of the children.

The classroom discussion was kept general and did not touch on what computer programs were, or the types of people who make computer programs, so as to minimise the influence on the pictures the children were about to draw.

In response to the opening question the younger children started by talking about the physical components of a computer.

> *It's like a TV. It's got a keyboard.*

They then listed activities computers can be used for.

> *You can go on the Internet. You can play games.*
>
> *You can write on (it).*
>
> *You can publish stories.*
>
> *You can play CDs on it.*
>
> *You can send email.*[2]

Similarly, the older children described a computer in terms of what a person uses a computer for.

> *You have games on it.*
>
> *You can play music on it.*
>
> *You can download things off the internet.*
>
> *You can send emails to people.*

The older children also tried to categorise computers and describe them in more general terms.

> *It's a thing that holds work and writing.*
>
> *It is a form of technology.*
>
> *It's a machine.*
>
> *It stores information.*
>
> *A computer is something you find information (in).*
>
> *It is a mechanism with an encyclopaedia in it.*

A child in the younger class did mention that a computer is a "thinking machine". Similarly it was mentioned in the older class that a computer "has a brain of its own – a metal brain".

A noticeable difference between the age groups was that the older children continually referred to computers as being sources of information for school projects and homework. This was also shown in the interviews. The younger children seemed less conscious of this and were happier talking about computers as something to play games on.

---

[2] None of the year 2/3 children knew what the "e" in email stood for, but they did know what email was. One year 5/6 child said it stood for "electrical".

After this general discussion, the children were asked to draw a picture showing a person programming a computer. No explanation was provided of what "programming a computer" meant. Many of the children (especially in the younger class) immediately said that they did not know what was meant. They were told that if they did not know what "programming a computer" was, they should draw a picture of a person using a computer. In both cases, they were told to bear in mind three things: what type of room is the person working in, what type of clothes is the person wearing, and whether there are other people in the room.

The hope was that the clothes and rooms would portray more of the feelings of the children towards computer programmers or users, and the number of people drawn would indicate something of the social aspects of the children's view of working with computers.

The drawings

The drawings were very useful in gathering information about the children's understandings. They showed what the children thought and possibly felt about a variety of topics concerning computers – what people do with computers, the environments where computers are used, whether or not people work on computers by themselves or in groups. The older children also attempted to convey something about their understanding of computer programming.

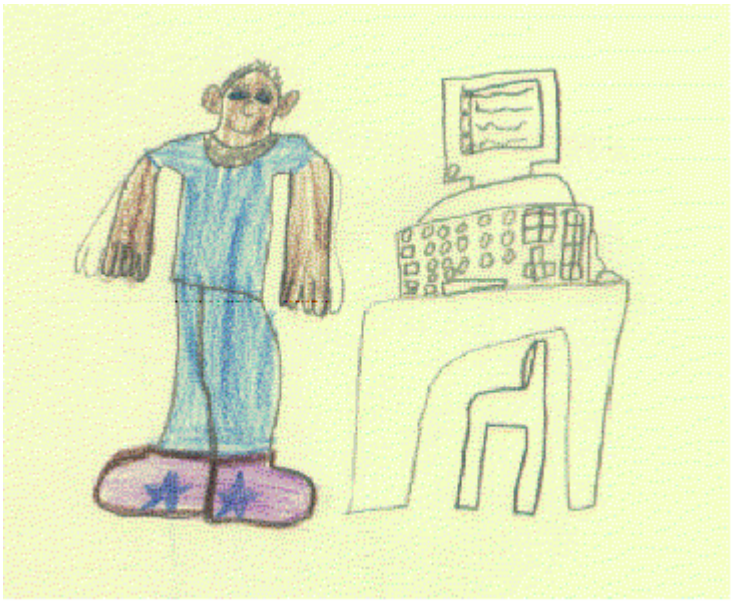


Figure 15 – A detailed computer showing lines of text.

As they had reported, the younger children had little idea of what computer programming was. Several pictures had squiggly lines on the computer screen to represent lines of text (this was a common visual cliché (Figure 15)). In the follow-up interviews, only one child said the person using a computer displaying lines of text was actually programming the computer. The others said the people in their pictures were writing stories or letters. The next most common activity depicted was playing games. Other activities were using the Internet: one screen showed information about pop groups and two of the children reported in the interviews that their pictures showed people "using the net" or "looking up things".

The drawings by the older children were remarkably different about what was visible on the screen. Seven of the fifteen drawings showed either a Windows start up screen or a message stating "Windows XX setup", where the XX was 95, 98 or ME (e.g., Figure 16). It is possible that some of the children were influenced by what they saw others drawing, but they were working at their own desks and there was not a lot of movement around the room. Some of the interviews indicated that this was an attempt to portray an understanding of programming. They knew it meant something technical and not just playing games and so they showed something that seemed technical to them. The only things they had experience of as being of this more technical nature were the start-up screens mentioning the operating system, interface items (such as menus) and installing programs. Installing programs was regarded as the same thing as programming in several of the interviews.

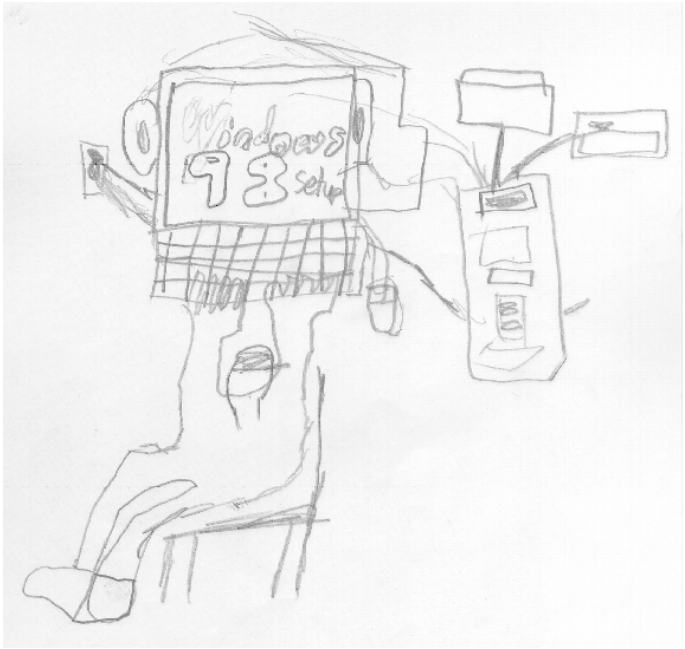


Figure 16 – A Windows' setup screen and a variety of peripherals.

The rooms

Even though the children were instructed to include details of the room the person and computer were in, most of them did not. This was true for both age groups. In one of the interviews when asked, "What type of room is it?", a 6-year-old boy replied, "I'm not sure; I just drew the computer; I didn't think about the room it would be in." A sizeable minority did include some office type furniture in their drawings, in particular wheeled furniture, especially office chairs. Of course, many homes include such wheeled chairs and computer desks. In the interviews, some of the pictures showing office chairs were identified as being in rooms at home. During the interviews, the children were explicitly asked to identify the locations of their drawings. The answers are summarised in Table 1.

The majority of drawings therefore appeared to show a room in the home environment, followed by an office of some sort. Barba found that older children portrayed an overall "office" atmosphere in their drawings (Barba, 1990). This seems to have diminished somewhat due to the large number of computers now in homes. There was no noticeable difference between the age groups or between

genders. One ten year old girl did however say that the computer was in her brother's bedroom, "He uses it for a study. It was downstairs but he wanted it in his bedroom."

So it appears that the children were directly applying knowledge from their home situations to the task of drawing someone programming (or using) a computer.

Location	**Count**
Office/Home office	7
Bedroom	6
Living/family room	3
Dining room/Kitchen	2
Computer room	2
Classroom	2
Study	1
Laboratory	1
Other	2

Table 1 - Locations of the computers.

The computers

There were further age related differences visible in the drawings. The drawings of the older children tended to be more technically correct. This agrees with earlier work that shows the expected relationship between age and an increasingly sophisticated understanding of computers (Barba, 1990; Denham, 1993).

A noticeable example of this was the number of drawings that portrayed mice attached to the computer. All drawings from the older group showed mice, but less than a third of the younger children included mice in their drawings. This is particularly interesting since the younger children could be assumed to use mice relatively more often than the keyboard, when compared to the older children, with their correspondingly higher levels of literacy. Similarly, the case for the computer as a separate unit from the monitor did not appear in any of the younger children's drawings, whereas a separate box was drawn by more than a third of the older children. The school had recently purchased Apple Imac computers; these include the processor, hard disk and other basic components within the case holding the monitor. The younger children were only used to seeing these computers in the classroom, whereas the older children would have experienced different computer configurations during earlier years at school. The fact that most children had computers at home, of which the majority were not Imacs, did not have as much influence on the younger children. It must be remembered that the drawings were produced in classrooms that had the Imacs at the back of the room. In this case, at least a third of the older children were using their memories, rather than referring to the computer in the room for their representations.

Some of the older children drew a large variety of computer peripherals including scanners, CD players, joysticks, speakers, cameras and printers (Figure 16). The only peripherals apart from monitors, keyboards and mice drawn by the younger children were CD players, printers and speakers.

Overall, the older children appeared to be more aware of the physical attributes of computers and the peripherals attached to them.

People

Some of the drawings showed the computer dominating the environment; others had very small computers with people and furniture being far more significant. Except for one drawing, all people were shown in normal casual dress. This could largely be the result of people being drawn at home, but it reinforces the image that using computers is part of normal everyday life. Even when children said the person in a picture was a computer programmer, the clothing was casual. The one exception (Figure 17) showed a programmer wearing a white coat as in a laboratory.

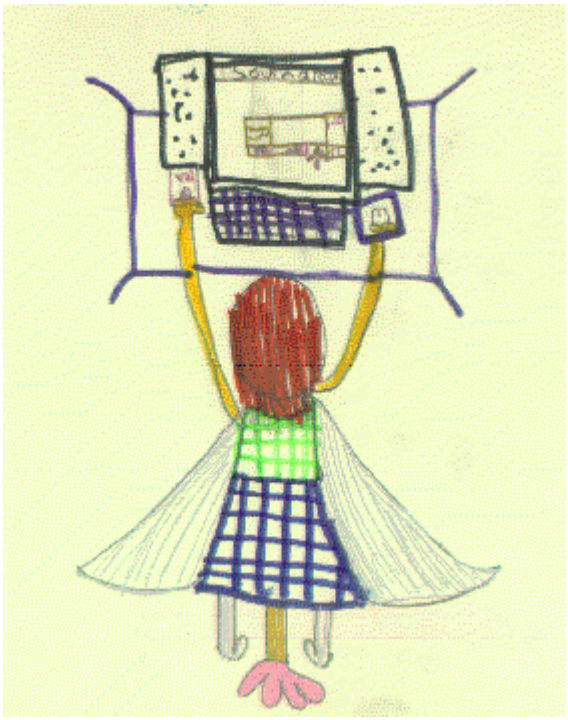


Figure 17 – A programmer in a lab coat.

Generally, girls drew girls or women using the computers and boys drew boys or men. Occasionally a child drew a person of the other sex or drew people of both sexes using the computer together. Sometimes it was difficult to tell from the drawing, the sex of the people portrayed. In the follow up interviews, some of the children had difficulty too. One ten year old girl when reporting on the person in her drawing said, "He is using it to do her work".

The number of people in the drawings was interesting. Most children drew only one person. While not statistically significant, the younger children, in particular girls, drew more people than the older children.

Interview Section

The classroom teacher of the younger children helped to choose a group of six girls and six boys from the first section of the study to take part in the interviews. The children were chosen to have a

mix of abilities. All fifteen children with participation approval from the older class were interviewed. This meant the older class provided six girls and nine boys. The interviews were audio taped and then transcribed.

The questions asked included:

Questions about computer use

How often do you use a computer?

Do you have a computer at home?

What computer programs do you use? What games do you like?

Questions about computer programs

What is a computer program?

Have you ever programmed a computer?

Have you ever wondered how computer programs are made?

Where do you find computer programs?

How do computer programs get into computers?

What is a computer program bug?

Questions about computer programmers

Explanatory questions based on the child's drawing from the classroom session.

How does a person become a computer programmer?

What do you imagine computer programmers are like?

What skills does a computer programmer need?

Do computer programmers work alone or in groups?

Questions about the child's interest in programming computers

Would you like to be able to program computers? Why? Why not?

Do you think you could learn how to program computers?

What type of program would you write if you could?

Do you like computers? How much?

A question I wish I had included was, "Do you know any people who can program computers?" Given the number of children who considered programming to be any sort of technical activity such as installing a program, I expect that many parents and older siblings would have been mentioned as people who could program. Some of the answers to other questions included this information.

Twenty-five of the twenty-seven children interviewed had computers at home and the children used them regularly (on average several times a week). There was little difference in usage between girls and boys.

The children responded with a wide range of programs and activities to the question, "What computer programs do you use?" Figure 18 summarises the different activities reported by the children. The percentages represent how many times that activity or a related program was mentioned out of all activities or programs reported by children from that age group. The results are consistent

with similar questions from another study (Van Duuren, Dossett, & Robinson, 1998). The older children reported using the Internet more than twice as frequently as the younger children did. The increase in use of the Internet came at the expense of games and education/edutainment programs.

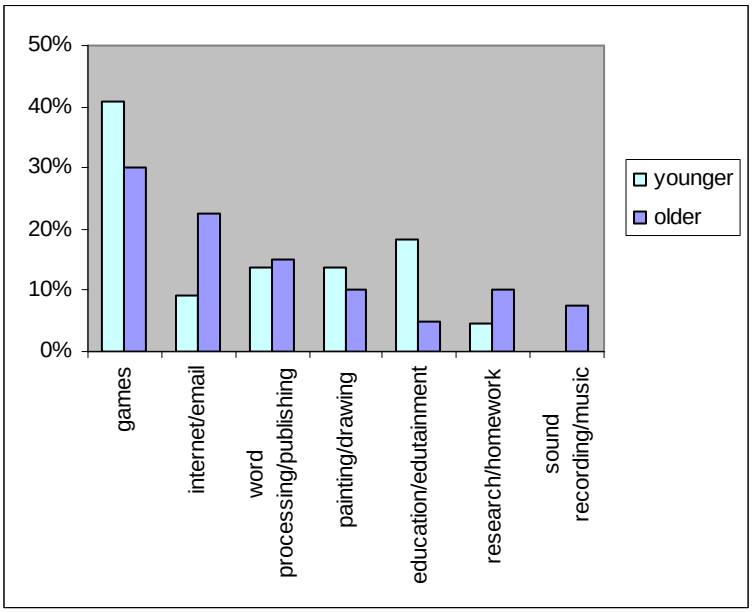


Figure 18 – Percentages of reported usage of programs.

It is generally true that children feel attracted to computers and enjoy working or playing with them (Papert, 1993). The younger children all reported a great liking of computers, as did most of the older children. Some of the older children expressed uncertainty as to whether they really liked computers. This reluctance was seen in both boys and girls. One ten-year-old boy reported, "I don't really really like them I just like to go on them often." A girl, after earlier stating that she did not like computers, said, "It depends if I'm playing a game on the computer, or if I'm using it for something totally boring like a topic for school. So if I'm using it like that, I would not like computers."

Computer programs

After describing the computer programs they use, the children were asked to explain what computer programs are. They were told to imagine they were talking to a child who had seldom used a computer, and asked what they would say to explain what a computer program is. This was a hard task. None of the children answered the question in the way a computer scientist would. There was no talk of instructions and control of the computer. Almost half of the younger children said that they did not know what a computer program was, even though they had happily told me what programs they use on computers only moments before. Earlier research has shown that even quite old children, up to eleven years old, answer this question with examples of specific computer programs (Van Duuren et al., 1998). The same study, demonstrated that children eight years old and younger do not understand the concept of programming something, such as a robot, to control its behaviour.

Even adults find this question difficult as we saw in Chapter 1.

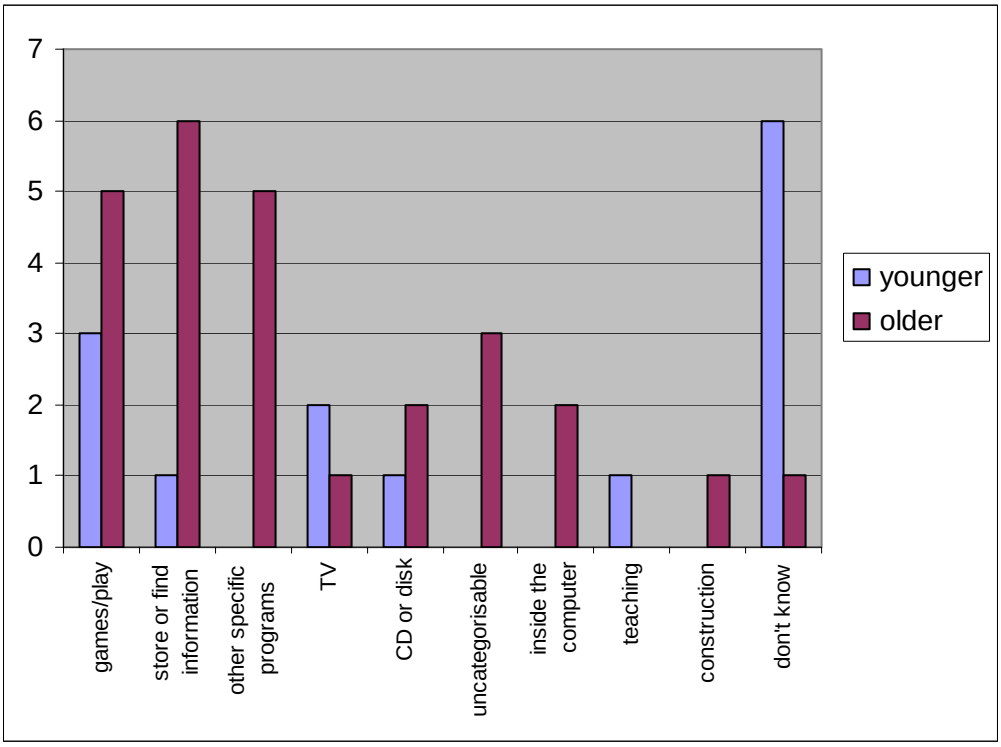


Figure 19 – Topics used to explain computer programs.

Figure 19 shows a categorisation of the topics the children mentioned when describing computer programs. Most of the children who responded with an answer apart from "I don't know", included an understanding of user-controlled activity when trying to describe what a program is. In this category were playing games and finding information. Language such as, "You can do things on it", and, "You can draw. You can play games", was very common. Programs were seen as things on the computer the child worked with to get things done. This is actually a very good definition of what programs are. Several of the older children mentioned that programs were a source of information or place to put things. Files were also mentioned as being synonymous with programs.

While the results here broadly agree with the earlier findings, we can see that by the age of ten many children are developing an understanding of some thing that is inside the computer that controls the interactions with the user. The younger children who attempted to answer the question were also aware of programs as allowing the user to do something with the computer.

Progression of understanding

As children get older, their understanding of computer programs develops. Originally, there is a vague idea of the thing that allows different activities to be carried out on the computer. In children closer to ten, this is supplemented with the idea of something with information that is put into the computer, which eventually develops into an understanding of the thing that controls the computer. This last stage appears to come later than ten or eleven years old. All of these understandings are correct. The progression is from a surface-level description of what programs are through to an appreciation of the components that constitute a program and the effect they have on the computer.

Most children reported that they had never programmed a computer (Figure 20), but a small number of children, including four of the boys in the older class said that they had. When questioned further about this the children mentioned a variety of computer tasks ranging from, connecting to the Internet, to installing programs, as what they meant by programming, which certainly matched the drawings they had produced. One ten-year-old boy reported, "I don't really read the manual but I help Dad to load them on", and a ten year old girl said, "Yes. It was when I was seven or something and my brother taught me. He taught me how to open up and to do art and music. I'm used to it now." None of the children had ever used a computer programming language.

There were some imaginative answers when asked how computer programs are made. Many answers made reference to the production of other media, especially TV programmes. Producing artwork, photos and sounds appeared frequently in the answers, along with the notion of collecting resources and tying them together. Other children focused on the physical aspects of a computer in order to explain the production of programs. They mentioned batteries, wires and chips. Given the children's ages, it is not surprising that they concentrated on aspects of programs that can be perceived, particularly pictures and sound. There was no indication of knowing how these things could be tied together and controlled.

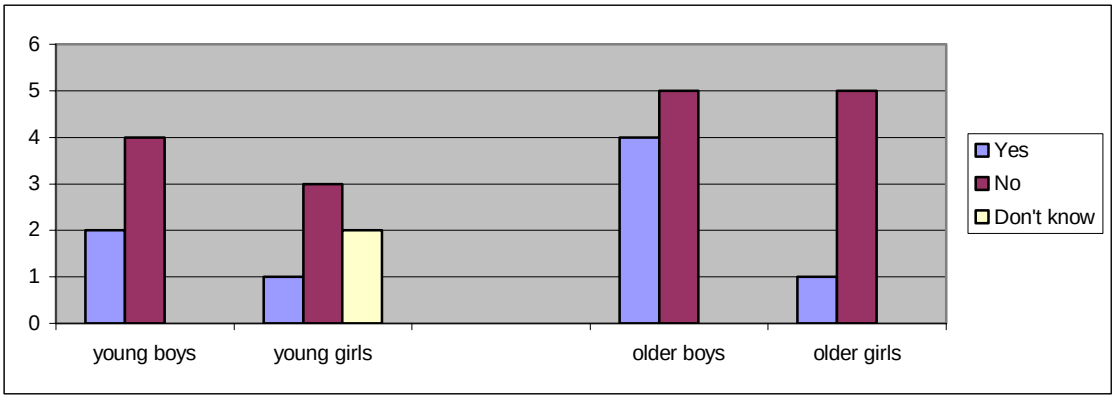


Figure 20 – Have you ever programmed a computer?

Some of the answers were plausible: "First of all they would have to get an idea of what they would want to make. Probably have to design it and that." Some knew how the big companies do it: "First they make it from a programme on TV then they make a CD and then they check it out if it works and if it doesn't work they just check it again just in case it might work."

There was no noticeable difference between the answers of the two age groups, but almost half the children could not give an answer, even when they said that they sometimes wondered how computer programs were made. Most children knew that programs came on CDs and some knew you could get programs from the Internet. Several also knew how to install programs from CDs.

Many of the children had heard of computer program bugs, but when asked to explain what a bug was they described viruses. Several children mentioned the "Love bug" virus, which had been widely discussed in the news media earlier in the year. No child talked about bugs as mistakes in programs,

but several children mentioned hardware not working or losing files. One older boy mentioned the Y2K bug as well as viruses.

Computer programmers

The children were also asked questions about computer programmers; in particular, they were asked what they think computer programmers are like and what skills a person would need in order to program computers. Once again, the younger children were less likely to be able to answer the questions.

It was quite clear to the children, how someone becomes a computer programmer: you have to learn about computers. They described a large variety of ways in which this learning could take place. Colleges, universities, and libraries figured widely, but other means included email, magazines and computer experts. One child optimistically said, "… they look in help. And the computer tells you what to do and things like that." Getting a lot of experience working with computers was also seen as a way to learn computer programming.

According to the children, computer programmers are very interested in computers and know a lot about them, including "secret stuff". They are intelligent, and similar to scientists. They are young and busy. They make a lot of money. They enjoy spending a lot of time using computers. The children regarded them as normal and nice. They like games. One ten year old girl obviously thought they would have to be very fastidious in their job because she said, "A bit like my dad. My dad explains things very, very boringly long."

The children described a large range of skills computer programmers need to have. They have to be creative and imaginative, very good at art, good at making things. They are also good at typing and moving the mouse. They have to be able to communicate well. My favourite skill was that they have to "know how to get the bugs out of it that they have accidentally put in."

Very few of the children thought that programmers worked alone all of the time. This contradicted evidence from the drawings produced by the children, but the pictures did not necessarily show programmers at work. Some thought that programmers worked alone, some of the time, and some of the time in groups. A small number of these mentioned that as a programmer got better, he or she would be more likely to work alone. This seems to indicate a belief that working in a group is most beneficial to less experienced or less able people.

Another aspect of working in a group, described by the children, was of people with different skills, such as drawing, working with those whose skills were putting things into computers. This view, whereby all people involved with the production of computer programs are regarded as programmers, appeared to be widespread. One girl mentioned that it must take about twenty programmers to create an educational program because she had seen the list of credits associated with similar programs and took all the names to be programmers. Therefore, programming involves any activity that contributes to the production of a program.

There was certainly no negativity expressed towards programmers in the sense portrayed in films and television, as being out of touch with the rest of the world. Of course, the children being interviewed did know that I was a computer scientist.

Would you like to program?

The younger children were enthusiastic about learning how to program (Figure 21). Both boys and girls in the younger age group were similarly keen to learn how to program. The older boys were also interested. However, only one of the older girls replied that she definitely would like to learn how to program. Some of this can be explained by the children misunderstanding the question. Several of them took the question, "Would you like to be able to program computers?" to be, "Would you like to program computers as your job?" In fact, one girl who responded uncertainly to the question said, "a little bit, because I am not fascinated on money". Whereas one of the boys said, "Yes, because I'd get paid a lot." When I inquired further and asked him if he would like to program computers even if it was not his job, he replied, "Yes, because I'd get to do something that is not boring."

It seems that some children become alienated from computers as they grow older, in a similar way to children who express less interest in science as they progress through school. Children in the older group were slightly more reluctant to express whole-hearted interest in programming. Several studies (Barba, 1990; Brosnan, 1999) show that girls seem to become less positive towards computers as they progress through primary schools. This trend appears here, when it comes to learning how to program.

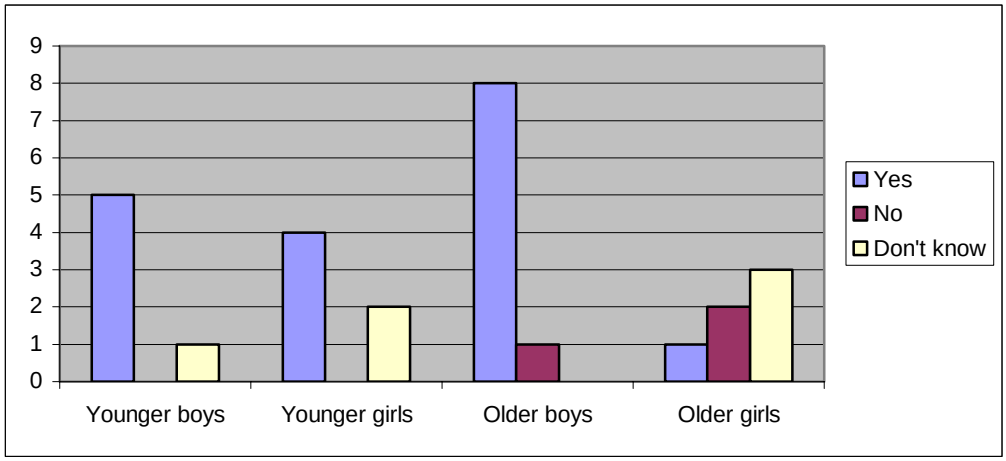


Figure 21 – Would you like to be able to program computers?

Regardless of whether a child was interested in learning how to program or not, almost all of the children said they thought they could learn how to program. Therefore, I asked them what types of programs they would make if they could (Figure 22). The vast majority of suggestions were games of various types: maze games, racing games, exploration games, learning games. Some children also suggested research programs such as encyclopaedias, drawing and writing programs, and programs associated with the Internet. One girl had years earlier designed a game to run on computers. It entailed a little girl moving around a world and answering questions. She made the game on paper because she did not know how to put it into a computer.

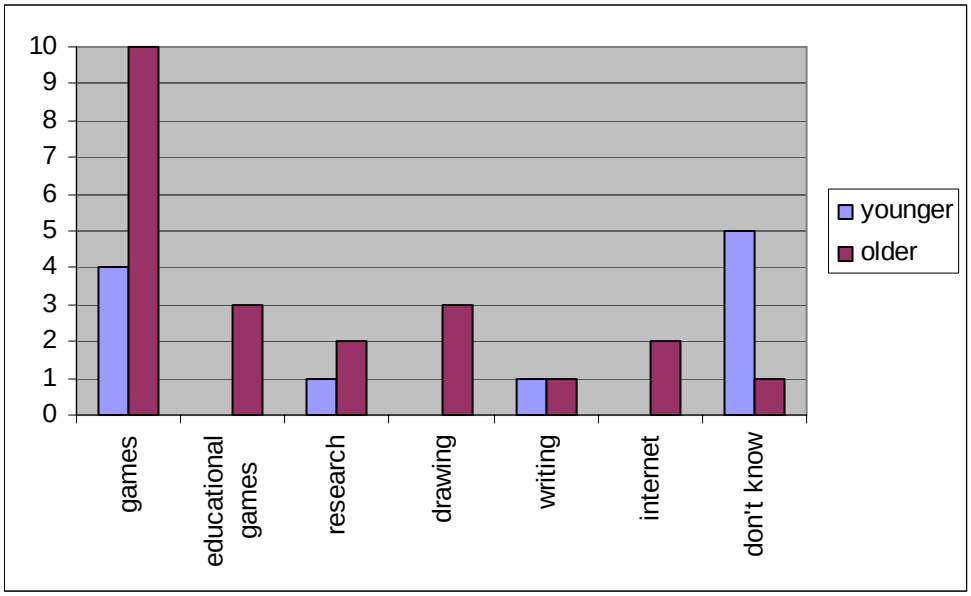


Figure 22 – What type of program would you make if you could?

All of the suggestions for programs were based on the types of program children had already seen or used. Several children did say they wanted to develop new games or new ways of doing things, but the only examples given were variations of software they had already experienced. This matches the results of Smith and Grant, when they ran a class for children to construct their own computer games (G. G. Smith & Grant, 2000); unless children are given guidance to develop something new they will copy something they already know.

To summarize the data collection aspect of the study, children in the six to ten-year age groups are very comfortable with computer technology and use it frequently. They can provide lists of computer programs they use, but find it very difficult to define what computer programs are. Certainly, they understand that computer programs are used to do things and they associate the concepts of action and interaction with programs. They have very little understanding of how computer programs are produced and usually do not think about it. Some children consider collecting static and moving images and sound, and tying them together, to be programming. The mechanics of how text, numbers, images and sounds are manipulated by computers, under the control of programs, are not understood or even guessed at by most of them. Understandably, children seem to comprehend computer programming as an extension of their own experiences using existing computer programs.

They do not have negative views of computer programmers and many of them would like to be able to program computers themselves. Generally, computer programming is seen by them as part of a collaborative process, with teams of people working together. They express strong interest in producing their own programs, usually games.

DESIGNING PROGRAMMING ENVIRONMENTS FOR CHILDREN

Two different approaches can be taken if we want to introduce children in the 6 to 10 age range to programming. Programming can be presented as something completely new. There are new concepts to understand, new ways of thinking required, and new terminology to learn. On the other hand,

programming can be built on the understandings children have constructed, during the hours they have spent using and playing with computers.

The first approach can certainly be argued for. Programming is different from almost anything else, although there are similarities with preparing lists of instructions for people to follow, such as giving directions to get to a person's home or providing a recipe for a cake. Even so, in several ways programming is different from providing lists of instruction in everyday life: the level of description required, the explicit control structures and the very particular syntax rules.

The second approach uses the experience children already have from using computers in order to introduce them to programming. Even though we have seen that not all the understandings children naturally construct about computer programs will be accurate, they can provide a starting point. A suitable environment could lead children to better understandings.

The aim in using this approach is to provide a programming environment that has points of contact with the children's current understandings. In psychological terms the attempt is to place the environment within the Zone of Proximal Development of the children (Vygotsky, 1978). Remember, the ZPD is the situation where learners are most likely to learn because of their existing knowledge and capabilities. When children are in the ZPD, they can perform tasks when interacting with others, which they could not perform if working alone. In the case of a programming environment, a major principle in the design would be to foster interactions that lead the children to discover more about programming, hopefully correcting misunderstandings along the way.

What are some of the ideas we could base such an environment on given the understandings revealed by this study? I believe there are four main points that can be supported by the findings:

1. Make the production and collection of resources such as pictures, animations and sound part of the programming environment.

The children considered the production of these resources a part of the programming process. If we couple this with the children's desire to make programs that resembled existing programs, it becomes essential that drawing, animation and sound effects are easy to produce within programming environments for children, or at least easy to incorporate into the environment. Some may argue that this requirement has nothing to do with programming. At the least, the inclusion of pictures, movement and sound makes the environment more enjoyable for children. If you look again at Figure 18 which shows the programs children use, it should be clear that these programs require these facilities.

2. Provide high-level instructions that correspond to the things children want to represent in their programs.

The children saw programs from a surface-level perspective. To them, programs were made up of those things that could be seen or heard. Thus, if we want these children to program, it makes sense to include program instructions that map closely to changes that can be perceived by a child when the program is running. It must be stressed that this is only a starting point. Eventually, as the programming projects undertaken by the children get more complicated, they will need to move to

more abstract computational mechanisms than simply moving characters around the screen or making a noise occur when the user presses the mouse button.

At the beginning however, if a child wants to write a storybook program there should be page placement tools and page turning actions available to be used in the program. Similarly if a child wants to make a game where an onscreen object changes shape and position in accordance with the actions of the game player, the instruction set should include something clearly associated with changing shape and position. This raises a dilemma that has affected visual programming and other end-user programming language systems for a long time – in order to provide functions close to the child's intended actions the system has to become very task-specific. The more task-specific an environment becomes the less it is able to deal with tasks outside that environment.

It appears from this study that when children first start to program they will want to produce programs that come from a rather narrow range of program types. This narrowness can be a benefit when designing a programming environment. It makes it easier to provide the task-specific tools the children will need to produce their early programs. This does not mean that children should be restricted to certain types of program, but that any programming environment for children should make it easy to produce the programs that children are most interested in constructing.

There are already some examples of this type of specialisation. The Stagecast Creator environment which started life as KidSim (D. C. Smith et al., 1994) was proposed as being PacMan equivalent rather than Turing equivalent. In other words, children should be able to produce programs at least of the complexity of PacMan.

Not all children want to write game programs and even those who do may want to write other types of programs as well. This means that the environment should not be restricted to games alone. A consequence of this is that a programming environment for children needs to provide a large number of high-level functions that are applicable in several different domains, e.g., story writing, picture drawing, game playing, and internet connectivity.

An alternative to one monolithic programming language with all of these capabilities is to provide an underlying language system, on top of which more task-specific functions can be built. Preferably, the underlying language would also be accessible to non-expert users.

3. Provide a readily comprehensible progression from the visible aspects of a program to the underlying rules or conditions that produce the behaviour of the program.

The children in this study showed very little understanding of what goes on beneath the surface of programs. To understand computer programs there are many things that need to be learnt, two of the most fundamental being the relationship between each instruction and its effect, and the importance of the order of execution of the instructions. The conditions under which the instructions are executed and the order in which they are executed determine the overall behaviour of the program.

The relationship between instructions and their behaviours has been partially dealt with in the previous recommendation. If the instructions produce desired results, they will be used. If the representations of the instructions map closely to the desired behaviours, it will be easier for children

to see the connection between the instructions and their results. This representation could be a clear textual description of the instruction or a picture or animation showing the effect of the instruction. For younger children a picture or animation would be preferable.

Not all instructions can be represented easily by a picture or animation; obvious examples include arithmetic operations and the assignment of values to variables. There are different ways children can be assisted in comprehending these lower-level instructions. One way is to make variables and their values visible. Some variables can be represented effectively by graphical means and changes in the variables can be made obvious. For example, a variable representing a colour can be shown as a rectangle filled with the corresponding colour value. Changes to the colour variable would be visible as changes to the rectangle. Similarly, an arrow can show the value of a variable representing direction. A slider control, or a bar that changes length as the variable changes value, could be a more general graphical representation of a numeric variable.

Support should also be given to help the children understand when particular instructions will be executed. On the surface, a list of instructions seems to convey the order in which the instructions will be performed, but in practice, most textual programming languages follow a more complicated execution sequence, starting at a main method and jumping between methods, only executing sequential lines of code in small sections of the full program. A rule-based language would be easier to follow in the sense of being able to identify when a particular sequence of instructions would be executed.

4. Make the environment usable by children when they are most interested in making their own programs.

The best time to teach anything is when the learners are most interested in the subject. Even in this small study, there was a decrease in enthusiasm towards learning about computers and programming as the children got older, especially amongst the girls. A simple, enjoyable programming environment the children could use to make their own programs around the age of eight might maintain interest.

The interesting thing about learning how to program is that the purpose is not really about programming. Learning how to program for its own sake is a largely meaningless task, whereas learning how to program in order to be able to do something that is relevant to the learner can be highly motivating. This ties back in to the second recommendation, and the first guideline from Chapter 1; the environment should make it easy for children to construct programs of the type they find interesting.

Other points

There are other aspects of the design of a computer environment for children, which are only lightly touched on by this study.

- The environment should be usable in a group situation.

There is a lot of evidence outside this study that demonstrates this is a good idea. Certainly, Vygotsky believed that children learnt in the context of social interaction.

- It should be easy to share and modify existing programs.

The children reported that programmers work in teams. By itself, this does not mean the design has to include collaboration at the technology level, as in Computer Supported Cooperative Work terms, but the sharing of ideas and activities should be encouraged by the programming environment. Children learn a lot from the work of other children.

SUMMARY

This study reinforced the widely held belief that children love computers. They have definite views on computer programs and the types of programs they would produce themselves if they were able to. They have limited understanding, especially those children under eight, of what computer programs actually are and how they are produced. Children under eight are very keen to learn how to program. Older children start to lose that enthusiasm.

Even the limited understandings that children have about computer programming can be used as a starting point to design a programming environment for children. By building on their current understandings, they should be able to use such an environment with less initial effort. Careful design of the environment could then lead them to both a deeper understanding of programming and provide them with the ability to produce a wide range of useful and meaningful programs.

From the four main points described above, I added three extra guidelines to the list given in Chapter 1. The third point was already covered satisfactorily by guidelines 4 and 7.

A programming environment for children should …

10. *include the ability to produce and collect resources such as pictures, animations and sound.*
11. *provide high-level instructions that correspond to the things children want to represent in their programs.*
12. *be usable by children when they are most interested in making their own programs.*

THE THEMES AND DESIGN OF ICICLE

This chapter considers the influences on and rationale behind the design of Icicle[1] (Interaction computing in a constructionist learning environment) (Sheehan, 2002). Chapter 5 describes the effect of this design from the point of view of someone using the system. Readers who prefer examples in order to understand concepts are advised to read Chapter 5 before reading this chapter.

GUIDELINES AND LESSONS

The guidelines from Chapter 1 and Chapter 3 are just that – guidelines. They do not specify exactly what a programming environment for children should look like and how it should work. They do offer help when choices need to be made as to how something should work or appear. Many possible environments can be produced in accordance with the guidelines. In order to produce a specific programming environment for children, some ideas for a computational model, and how it could be represented to the children, needed to be found.

The lessons learnt from ICE in Chapter 2 were good starting points. Another starting point was another programming environment for children, Stagecast Creator. We will examine Creator and note some of its limitations. Several themes will be developed based on the lessons from ICE and as solutions to the noted limitations of Creator. These themes form the basis for the design of Icicle.

The guidelines are collected here because they will be referred to frequently.

A programming environment for children should …

1. *be able to produce programs of a wide variety of types, especially the types that children most want to produce.*
2. *give the children objects they can see and manipulate to program with.*
3. *allow the objects to be manipulated in a series of small reversible steps.*
4. *make the sequences of instructions in a program explicit and easy to follow.*
5. *be interactive or lively.*
6. *provide a variety of ways of using the environment, starting with a very simple entry level and a gentle learning curve.*
7. *make its models explicit.*
8. *be fun.*
9. *not be ready-made.*

[1] Originally it was ICICLE., and it stood for Incremental Computing In a Constructionist Learning Environment, stemming from its relationship to ICE. The acronym was changed to signify interaction as the most important concept and the case of the letters was changed because it is easier for children to read words in lower-case.

10. *include the ability to produce and collect resources such as pictures, animations and sound.*
11. *provide high-level instructions that correspond to the things children want to represent in their programs.*
12. *be usable by children when they are most interested in making their own programs.*

The lessons from ICE were:

A. The low floor approach.

B. Display source code as it is being produced.

C. Relative positioning of objects is more powerful than absolute.

D. Source code should be readily understandable even to a novice user.

E. Make interaction possible.

STAGECAST CREATOR

By the 1990s, twenty or so years after Logo was developed, fundamental changes had occurred in the computer world. The speed and size of processors and memory had increased rapidly, but from the user's point of view the biggest change was moving from a keyboard-centric command line approach to control the computer, to a pointer-centric (usually a mouse) graphical user interface (GUI) that used direct manipulation of onscreen controls to achieve the same result.

For most simple tasks, this approach was easier for users, especially for novice users. Although many programs, including those for children, employed these techniques from early on, programming environments did not. The immediate reason for this was that programming languages were almost exclusively textual. The commands and instructions were entered as text and translated by compilers or interpreters into actions that the computer could perform. When visual programming languages (Chang, Ichikawa, & Ligomenides, 1986) began to develop, similar environments were produced for children. Apart from Logo, all of the environments mentioned in the introduction use direct manipulation of graphical user interfaces at least to some degree. Boxer (DiSessa & Abelson, 1986), Squeak (Ingalls et al., 1997) and Hands (Pane, Ratanamahatana, & Myers, 2001) are largely textual but require significant amounts of direct manipulation to use. AgentSheets (Repenning, 1993), ToonTalk (Kahn, 1996) and Stagecast Creator (D. C. Smith et al., 1994) are largely visual and programs can be produced solely by direct manipulation.

Stagecast Creator has a long history. Originally, there was a prototype called KidSim produced at Apple Computer under the direction of Allen Cypher and David Canfield Smith. The final version of this was known as Cocoa. It was eventually converted into Creator and released as a commercial product by the Stagecast company (D. C. Smith, Cypher, & Tesler, 2000). It is a rule-based system, meaning that all program behaviour is caused by matching rules with the current state of the program and executing the instructions of the matched rules. You can imagine a Creator program as a long list of *if some condition then do this* statements in a more traditional programming language. The *if* and *then* parts of the rule are known as before and after parts. The novel aspect of Creator is that rules are defined graphically, with the before and after parts represented as a series of squares with objects in

them. Because of this Creator is also known as a graphical rewrite system. Figure 23 shows a typical Creator rule. Objects or agents of the rules are called characters in Creator. In the before part, on the left-hand side, the boy character is beneath a bottle character. In the after part, the boy character has changed its appearance to be facing the other way with its hands extended and the bottle character has moved down and to the left. This rule is activated whenever a section of the Creator stage, the window that the characters move around in, matches the four squares on the left-hand side. This rule does not match if there is a character in either of the two empty left-most squares for example.

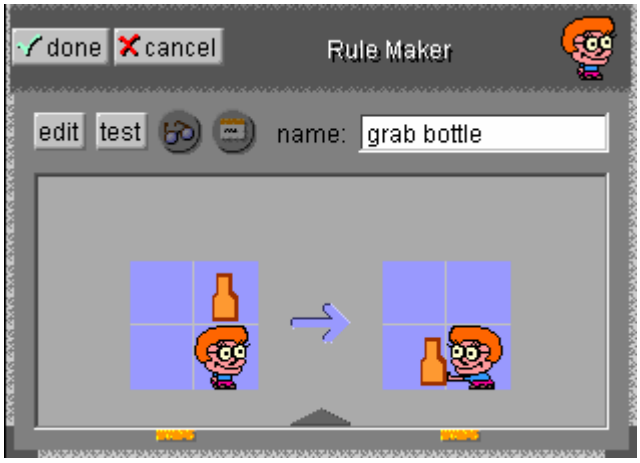


Figure 23 – The before and after conditions of a rule.

Creator programs run as a series of discrete clock ticks. In each clock tick the Creator system inspects the characters in the order they were added to the stage and looks through the before parts of the rules defined for each character. A rule is executed when a match is found between the before part of the rule, and the corresponding area of stage around the character. You can specify that more than one matching rule can fire for the same character if you wish. As the rules are executed serially until completion, if one rule for a character leaves the stage in a state that matches another rule for the same character, that other rule may also be executed in the same clock tick.

Programming in Creator

To make a program in Creator a child produces some characters and drops them onto the stage. To define a rule a section of the stage is selected, this brings up the Rule Maker window as in Figure 23. The child can then make changes to the after section of the rule.

Analogical representations

Smith et al. give two rationales for this approach to programming. They talk about Sloman's Fregean and analogical representations (Sloman, 1971). A Fregean representation is a generalized abstraction that can be used to signify relationships in a very compact manner, such as predicate calculus. Structure in the thing being represented is not necessarily obvious in the representation. An analogical representation is less general but more expressive. A map is an example of an analogical representation. Analogical representations are generally easier to work with than Fregean ones, but

less powerful. The trade-off with generality is acceptable in the intended domain of a programming environment for children.

Creator is obviously analogical. In fact, the representation of rules matches very closely to the effect the rules have when they are activated.

Iconic, Symbolic and Enactive

The other reason given for the programming approach of Creator is based on Jerome Bruner's description of the three ways knowledge can be represented (Bruner, 1966). Knowledge can be *enactive*, in the sense that we know how to do something, such as ride a bike, but it is difficult to learn how to do it using words or pictures. Knowledge can be *iconic*, in the sense that a picture can represent the information adequately. A picture of a bicycle can convey what a bicycle is. *Symbolic* knowledge uses a set of symbols and logical propositions to create and modify further propositions. Language and mathematical notations are symbolic.

Creator utilises all three forms of knowledge. The child drags characters around the screen, the rules show pictures of before and after states and there are variables that can be manipulated to provide greater power than the simple graphical rewrite rules.

Problems with Creator

Creator has been very successful. Children enjoy playing with it (Howland, Laffey, & Espinosa, 1997) and it is easy to produce simple programs with it. This does not mean that children necessarily understand how Creator programs work. Studies on KidSim and Creator, show that children do not comprehend the before condition of rules very well, especially with respect to the size of the stage area to include in the rule definitions (Gilmore, Pheasey, Underwood, & Underwood, 1995; Seals, Rosson, Carroll, Lewis, & Colson, 2002). In another study (Rader, Brand, & Lewis, 1997) it was shown that even after prolonged use many children had difficulty reading and understanding rules. There were several interesting observations.

Misunderstanding relative movement

Because of the left to right placement of the before and after sections of rules, children misunderstood the results of simple rules, e.g., in situations like Figure 24, many children reported the result of the rule was to move the character down and to the right, and in situations like Figure 25, many children could not see that the rule was to move the character to the left. These mistakes are similar to the orientation mistakes when turning the Logo turtle that were mentioned in Chapter 2. Both situations require the children to be able to imagine a frame of reference different from their own. In the case of Creator, the problem was possibly exacerbated by the right pointing arrow in the Rule Maker window.

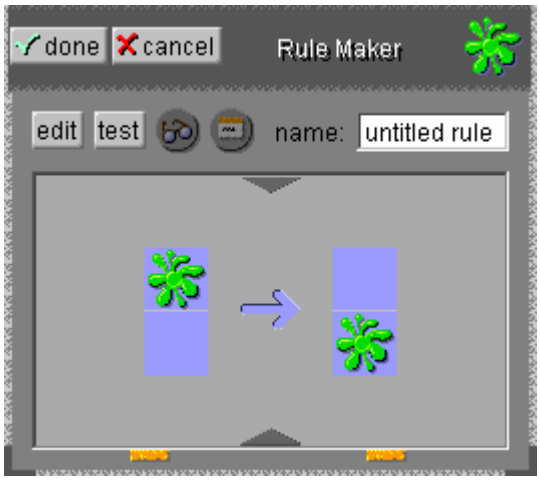


Figure 24 – A move down rule.

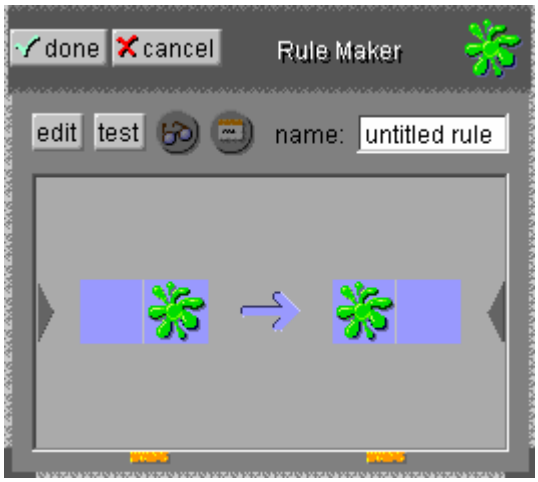


Figure 25 – A move left rule.

Order of rules

Which rule is executed depends on a variety of situations in Creator. Normally the first rule that matches for a character is the rule that works. This situation can be changed by grouping rules together and applying a different selection policy to the group, but children were not sure which rule would be selected from amongst all the matching rules even in the simple case.

Violation of local state

Another source of misunderstanding occurred when two or more characters appeared in the before part of a rule. In this situation, a rule associated with one character could cause another character to change its state. Since the action of a character can be determined by a rule in another character, it makes it difficult to ascertain why a character behaved as it did. Not only do all its rules have to be inspected but all the rules of all other characters as well.

Discrete not continuous

The implementation of Creator is discrete rather than continuous in several ways. Rules are scheduled one after another rather than in parallel, characters can only occupy positions in the stage that correspond to the underlying grid of squares, and the appearance of characters change abruptly from one representation to another.

Creator schedules rules in a discrete fashion. If two rules should both be executed in the next tick of the clock, Creator will completely carry out all of the instructions associated with one rule before performing any of the instructions of the other rule. This simplifies things in that the instructions for one rule do not interfere with those from another. If the first rule changes the state so that the second rule no longer matches then the second rule will not fire.

Even though these decisions are reasonable, they do limit the expressiveness of Creator.

Creator rules are based on a fixed two-dimensional grid. Most characters fit in exactly one square at a time. Larger characters can occupy several squares comprising a rectangle but they are still constrained to an area of the stage determined by the grid. Creator employs these discrete positions as

a way to simplify rule creation and definition but this means that all motion is stepwise, and movement at any angle other than to one of the other squares in the stage is impossible. Creator is not alone in this restriction, AgentSheets (Repenning, 1993; Repenning & Sumner, 1995), also puts objects into squares in a rectangular grid. These limitations have led to the development of another form of graphical rewrite rules (Harada & Potter, 2003).

The appearance of characters in Creator can be changed by setting the *appearance* variable. This way characters can appear to face in different directions e.g., Figure 26 shows four appearances for a train character. If users want to show the train turning gradually from one direction to another they need to create further appearances that would be shown between the appearances in the figure.

It may not be apparent to a user of Creator but there is no concept of direction associated with characters. Any understanding of direction employed by a user when programming in Creator must be associated with an appearance so that the system can distinguish the differences. In many situations, these limitations are not important, but in combination with the problem of one character altering the behaviour of another one, and the way Creator schedules rules they lessen the appearance of the characters as real objects.

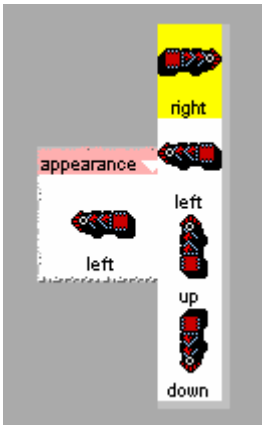


Figure 26 – Multiple appearances for a character.

Because the concept of heading for characters in Creator is implied by the use of different appearances, it means more rules have to be produced to deal with essentially the same situation, e.g., the train facing left being on the right of a section of empty track and the train facing right being on the left of a section of empty track. Thus it is common to get very large numbers of rules to deal with all scenarios (Seals et al., 2002).

THE THEMES

Four themes were extracted by reflecting on the lessons learnt from ICE and Creator: *lively concrete objects*, *interaction computing*, *closeness of mapping* and *ease of parallelism*. A fifth minor theme was added to make the environment usable to a wider range of children – *flexibility*. Each of these themes will be explained in detail but a brief overview will show how they came about.

Icicle was going to be a program to produce games and simulations. This meant it required objects similar to Creator's characters to represent the computations, but the characters in Creator do not

appear real because of the discreteness limitations and the violation of local state. You should be able to place characters anywhere in a program with any orientation and size. They should change smoothly from one shape to another rather than abruptly jumping from the current shape to a new one. I refer to these techniques to make the characters seem more real, as the theme of *lively concrete objects*. The objects should also react to changes immediately – hence the lively.

The terminology actually used in Icicle is to call the objects that control computation and move around the screen, *performers*, rather than *characters*, and the area of the screen that they move in, the *world*, rather than the *stage*. From here on, I will use this terminology.

The most important lesson learnt from ICE was the need to make interactions possible between the programs produced by the children and the people using the programs. This was reinforced in Chapter 3 when children responded that the programs they were most interested in producing were games. Games are interactive by nature; the person playing the game does something and the program responds to the action.

Interactions can also occur between objects in programs. An object can collide with another object and this collision can cause a change of behaviour. By generalising this concept to include all state change as a form of interaction, it became possible to produce a computing environment in which all instructions were based on and activated by interactions. I call this approach *interaction computing*. The interactions are used as the conditions in rules that specify the behaviour of the performers.

Closeness of mapping is one of the Cognitive Dimensions of Notations (Blackwell et al., 2001; Green & Petre, 1996) and is the measure of how closely a representation appears like the thing it represents. The source code in ICE did not represent the instructions in a way that was obvious to novice users. Using this dimension as a theme meant producing source code that appeared as much like its resulting actions as possible.

In Creator, choosing which rule would execute if several rules matched was difficult for children to understand. All matching rules should execute. In order to maintain the truthfulness of their starting conditions the rules must execute in parallel. As parallelism can also be difficult to understand another theme in the design of Icicle was to make parallelism simple – *ease of parallelism*.

Flexibility is related to the theme of *lively concrete objects*. It refers to being able to make changes at any time and being able to produce programs in a variety of ways, thereby making the environment accessible to different types of learners.

LIVELY CONCRETE OBJECTS

The desire to allow smooth, continuous change to Icicle performers and not to place limitations on position or orientation was motivated by the proposition that objects that appeared more real, lively or animate would assist the users to understand the computational process. In his thesis on programming with agents, Michael Travers argues that animacy is a basic category of the human mind and can be used to understand action and causality (Travers, 1996). Certainly, the Piagetian pre-operational child understands the world as being full of animate forces.

Since the performers would not be perceived as animate if they did not exhibit behaviours, Icicle makes the performers respond to conditions in the program. They respond to stimuli and are manipulable by the users. Their appearances, behaviours and other properties are changeable. Programs are represented as the behaviours of the performers and the users can modify programs by training the performers to learn new behaviours or changing the behaviours they currently have.

This fits in perfectly with the work on natural programming (Pane et al., 2001) where it was demonstrated that non-programmers normally attempt to assign behaviours to entities rather than to modify properties. This view of performers as realistic objects with their own behaviours is in accordance with guidelines 2, 5 and 7, that recommend *giving the children objects they can see and manipulate to program with, being interactive or lively* and *making models explicit*. Guideline 2 means that there must be a concrete way of representing the data elements in a program for the children to work with.

All mention of being more *real* or *realistic* has to be understood carefully. All shapes that move around the screen are real in the sense of being visible. Solid objects in the physical world have additional properties that we assume makes them real, e.g., continuously occupying space and behaving in accordance with physical laws. Animate objects are a specialised case of real objects. We assign animate objects properties associated with internal states. These internal states are understood to cause the behaviours the objects exhibit in response to external stimuli. In this sense, making performers *more real* means making them appear to have more of the properties of physical and animate real objects.

In Chapter 3 we saw that children want to produce games and most games can be represented by onscreen objects that move around the screen and interact with each other. Certainly all video games and board games match this model, as well as games that simulate sports or other physical activity. Even card games can be represented this way, even though it may not be the most natural mapping. If we then associate behaviours with the onscreen objects, rather than some other, hidden computational mechanism, we are providing a model that is both explicit and natural in the way that animate objects react to their environment and determine their own behaviours.

In practice, many games or simulations require multiple objects of the same type. The solution to this in Icicle is to associate each performer with a window, known as a performer box (see Chapter 5). The performer box encapsulates all the information about that type of performer, such as the possible shapes of the performers and the rules associated with them. Many performers can be made from the same performer box; they are all of the same type.

Guideline 5 is concerned with making the environment interactive or lively but the same can be said for the objects in the environment. Tanimoto defined four levels of liveness a programming editing environment can provide ((Tanimoto, 1990) as reported in (Burnett, Atwood, Djang, & Reichwein, 2001)). Level one provides no semantic feedback to the user as changes are made, level two provides feedback when directed, level three provides continuous feedback of the state modified by editing, and level four adds to level three, feedback provoked by any other changes that may be occurring, such as mouse clicks and time based events. Icicle achieves level four liveness. When a child modifies an object, it reacts immediately. This makes the object appear more realistic. Even

though it is strictly an unresolved research question (Blackwell, 2001), it seems plausible to many researchers that the more realistic objects appear, the easier it is to work with them and give them behaviours. Hence, the decision was made to make Icicle performers appear as real as possible, given the constraint of being simple two-dimensional shapes. Icicle uses several techniques to foster this notion of performers being real objects:

- shape production by manipulation
- shapes changing by morphing
- immediate updates
- immediate interruption
- no restrictions on position, orientation, or size, and smooth movement in any direction
- control of their own destinies

Shape production by manipulation

The performers require shapes to represent them on the screen. As Icicle is a prototype system, the decision was made to restrict each performer shape to one polygon, filled with one colour. This has an obvious effect on the potential detail of the shape, and will be commented on more fully in Chapter 7. Using simple shapes was influenced by guideline 9, that objects should not be ready-made. The children have to create their own shapes.

In an attempt to make the shape itself seem like a real object, it was decided to create the polygons by manipulating existing shapes. Different shapes are constructed by modifying a very simple base shape, effectively by squeezing it and stretching it as will be seen in Chapter 5. Icicle presents four purposefully bland base shapes: an octagon, a triangle, a square and a thin rectangle. Every performer is produced by modifying one of these shapes. The starting shapes are ready made, but they are abstract and the intention was that their abstract nature would encourage children to modify them, and turn them into shapes representing different objects. This was indeed shown to be the case when children used the system. The reason for providing base shapes rather than a blank shape producer, was partly due to guideline 2. The children are presented with objects they can see and manipulate. Four shapes are presented rather than one, such as a square, to assist the children if they want something more like an equilateral triangle than a circle for example.

The performers are the objects that move and react to events in the program. Sometimes the shape of the performer will have to change to represent a change in the state of the performer, e.g., a conductor performer must be able to move its arms in order to conduct other performers. This means that performers require more than one shape.

Name shape

If there is more than one shape for a type of performer, there must be a way of referring to the type that is unique. This is useful when a performer is referred to in a rule. As we shall see, the performers are represented in rules by a shape. To provide this unique identification, one of the shapes for each type of performer is known as the *name shape*.

The name shape exists partly because of guideline 12. If Icicle is to be used when children are most interested in making their own programs it needs to be at least partially usable by children around the age of eight. Because of this, almost all facilities of Icicle can be used without the child being able to read or write. Thus, performers are named by a shape and other objects, such as messages, are also named by graphical representations. The only names that need to be typed in with the current version of Icicle are the names for variables, saved worlds or sets of performers. There is nothing to stop pictures being used as the names in these last two situations, similar to the way Squeak allows projects to use default names and represents projects on the desktop as shrunken pictures of their contents. The naming of variables will be discussed later.

Even for children who are comfortable reading and typing, it can be helpful not to force the user to supply names for objects. Quite often, the names children give objects are chosen for convenience rather than descriptive purposes, as I found out when children were saving their own Icicle programs. Careful naming only becomes a meaningful requirement for users after some time with a system, when they have discovered for themselves that the convenient names have caused confusion.

Shapes changing by morphing

The onscreen appearance of the shapes is the principal way performers are perceived by users, and yet, as programs run, performers sometimes need to change the shapes they are using. One way to convey a sense of concreteness about each performer as it undergoes such a change is to make it transform its shape by morphing. A performer changes continuously from one shape to another, there is no abrupt change where the previous shape disappears and the new shape appears. This helps to maintain the sense of reality, that a shape *is* a performer with characteristics and behaviours.

Immediate updates

Another technique used in Icicle to make performers appear more real is to convey changes to performer appearances and behaviours even while a program is running. The world can keep running while new performers are created and added to it by the user, or while old performers are modified. This produces a sense of liveliness that hopefully makes the performers appear more like active agents in the world, as in LiveWorld (Travers, 1996).

As the performers in the world are representations of a type described in a performer box any changes to the performer box immediately affect all performers based on that box in the world.

Immediate interruption

Associated with liveliness is the need to handle interactions immediately. Interactions between performers can happen at any time. Similarly, the user can generate interactions at will, by a key press or moving a performer with the mouse. It is necessary to handle the interaction immediately or else the state that produced the interaction may no longer hold. Obvious examples include collisions between performers and key presses.

No restrictions on position, orientation or size, and smooth movement in any direction

Just as morphing from one shape to another encourages the perception of a performer as real, the smooth movement of performers in any direction does the same. Real objects can exist in any position and when moving from one position to another they occupy all positions in between. Except for some situations, such as teleporting a performer from one area of the screen to another, the motion of performers should be continuous without jumps. In Icicle, this is not literally true, the performers are restricted to the pixel positions of the display, and if a performer is moving quickly, it will not appear in all intermediate positions. Even so, in the majority of cases it looks as though the performers are moving rather than being erased and redrawn in a different position because the moves entail some overlap between the previous position of the performer and its next position. This flexibility adds significantly to the types of program that can be produced.

To permit this flexibility, rules could not be based on before and after pictures showing performers in discrete squares; another technique had to be found. One solution to this (Harada & Potter, 2003) employs a fuzzy concept of relative position and orientation in its rules. Icicle does not use relative positions of performers as the before parts of rules, instead it refers to specific interactions. The types of interactions and what an interaction is, will be discussed shortly.

Control of their own destinies

The last component of *lively concrete objects* is to do with ensuring that performers cannot be directly operated on by other objects. Even though real objects can be manipulated by other objects, e.g., a ball can be hit by a bat, things are made clearer if this is not allowed in a programming environment. In particular, if the behaviour of objects is localised, then finding the cause of the behaviour is easier. Not doing this was a problem with Creator. It also provides a consistent answer to the question of where should the result go. In the case of the ball being hit by the bat, what should the result be associated with, the ball, the bat or the hit? The decision made in Icicle is that a performer can only be modified by its own rules. Each performer is in control of its destiny. The result that moves the ball must therefore go with the ball. Because events, such as the ball being hit, are used to activate rules in Icicle this turns out to be a natural way of dealing with the problem.

Unfortunately, sometimes the simplest approach to a problem is to allow one object to manipulate another object. In cases like this, Icicle allows one performer to tell another performer that it should do something. The something is still carried out by the performer that is told.

INTERACTION COMPUTING

The choice of interactions as the basis for computing in Icicle means that the system is a rule-based system like Creator. Programs are represented by rules describing what to do when particular interactions occur. Icicle can be used to produce games and simulations and this means that the interactions have to be of the type required for these types of programs, e.g., key presses by the user and collisions of objects on the screen. The idea of interaction is generalised from this to mean any

change in state in the program; this even extends to variables. Assignment to a variable is an interaction. Different assignment values cause different rules to be executed.

How should the rules be produced within the Icicle environment? This is really two questions. The first is, how does the user inform the system of the conditions or interactions that activate the rules? The second is, how does the user enter the instructions associated with the rules? The first of these will be looked at here; the second is covered in the section on *closeness of mapping*.

When an Icicle program is running, interactions will occur. Rather than requiring the user to define the interaction when describing a rule, why not allow the system to request a rule when an interaction occurs that it does not have a corresponding rule for?

Meaningful interaction

With most programs that are executed directly by people, as opposed to programs run behind the scenes, the programs wait for some input from the user and then display changes in state. Usually this is repeated until the program run is finished. This is true whether the program is a spreadsheet, a word processor, a web-browser, a typing teaching program, or a flight simulator.

This cycle of input/output operations between the user and the program can be seen as an interaction between the program and the user. Typing with a word processor is interactive in this sense, but this is a weak definition of interactivity and does not take us very far towards a technique for creating programs. The information presented to the user by the program is only a representation of the text typed by the user. There is very little extra information provided by the program or guidance from the program, to help the user complete the task. Because of this, the user is not responding to the output provided by the program. There is no significant two-way communication occurring.

The interaction when playing a game or using a simulation is of a different kind. The actions of the user change the state of the program so that the program's behaviour is altered; it is not just adding extra letters onto the end of a document. Similarly, the altered behaviour of the program changes the following actions of the user. This is more like a conversation than the one-way storytelling of using a word processor.

In some programs, the interaction with the user can be understood as conveying meaning or intent to the program. An example could be a database query program that generates queries for the user by asking a series of questions. The replies enable the program to develop a model that can be executed (Baer, Groenewoud, Kapetanios, & Keuser, 2001). These interactions are meaningful to the user and, in the sense that the program generates behaviour from them, meaningful to the program. This type of question-response behaviour I call *meaningful interaction*.

Most programming environments are more like word processors than they are like games or simulations with the notable exception of PBD environments. Apart from Stagecast Creator there is another PBD environment developed specifically for children – ToonTalk (Kahn, 1996). (AgentSheets was not developed specifically for children.)

ToonTalk was designed to appear as a game with animated characters. Robots can be trained by giving them a box containing parameters. Then the user demonstrates what to do with the parameters. After being trained, the robots execute whenever they are presented with input that matches their parameter boxes. Parameter boxes can be generalized by removing values, e.g., removing a number from a box means the box can now match any value.

Neither Creator nor ToonTalk provide meaningful interactions in the sense of asking questions and producing behaviour from the replies. We have seen how a child produces rules in Creator. ToonTalk uses its game-like quality to make it easy for a child to teach robots how to behave but it is not easy to spot situations where the behaviour is incorrect and modify it.

The type of meaningful interaction intended here, must make it easy for a child to perceive a state that needs altering and then provide the tools to carry out the change, or define the behaviour. Gamut (McDaniel & Myers, 1997) allows a user to create and modify games in just this way. The user provides examples of behaviour either to teach the correct moves or to inform the system that some move is incorrect. The system fosters the appearance of conversation by asking the user to add more information when an action is ambiguous. The Gamut approach uses a sophisticated inference engine to deduce rules that match the examples but a simpler approach can be taken with less intelligence built-in to the system.

The two-way communication implied by meaningful interaction is important because it provides the user with a stimulus to make further decisions. This is especially helpful for beginners who do not know where to begin or what to do next when producing a program. In Icicle the system asks the user to define rules whenever a situation occurs which does not currently have an associated rule.

Interactions then become central to running Icicle in two related but different ways: the technique used to produce programs and the way in which instructions are scheduled. Icicle programs are developed using the interactions that occur between a child developer and the system and instructions are scheduled when interactions occur between objects or variables in an Icicle program or between an Icicle program and the person using it. The interactions convey the meaning between the developer and the environment, and can be seen as describing the program that is produced.

Programming by Rehearsal

The program production part of *interaction computing* is a form of Programming by Rehearsal (Finzer & Gould, 1984) or Programming by Prompting. Programming by Rehearsal gets the user to make choices when additional information is required. The choices are of a very simple type and the system does not make any inferences. Nevertheless, a dialogue between the environment and the programmer takes place that both prompts the user for information and changes the subsequent actions of the environment.

The program running and control part of *interaction computing* can be seen as a variant of event-based computing where a section of code is executed when some event occurs. Event-based computing gets its name from the GUI (Graphical User Interface) environments of the 1980s but is used in many situations such as simulation and hardware description languages e.g., Simula (Dahl, Myhrhaug, & Nygaard, 1970) and Verilog (Sandler, 1987).

Stimulus-response systems

Interaction computing can also be understood as a form of PBD stimulus-response system. Systems such as Pavlov (Wolber, 1997) and Gamut are categorised as stimulus-response systems, where the system is trained to produce a response when a specific stimulus occurs. In Pavlov, the user explicitly has to enter a separate mode to teach the stimulus; in Gamut, the separate mode is unnecessary because the user can show the stimulus using direct selection. In Icicle, a separate mode is also unnecessary because all events cause a match with a corresponding reaction. If a rule with a condition that matches the event is not found, then Icicle, either asks the user to define the required action, or it automatically creates a rule with a default empty action.

Production systems and non-programmers

In a study of non-programmers' solutions to programming problems (Pane et al., 2001) more than half of the statements used to describe the program were in the form of production rules. Creator, AgentSheets and Icicle can also be viewed as production systems (Davis & King, 1984). A production system is usually defined as a set of rules, a database and an interpreter for the rules. The interpreter scans the database trying to find conditions that match rules. There is commonly some decision as to which of the matches should be chosen. The chosen rule or rules are executed. The result of this is that the database is altered and the process begins again. In Chapter 6 we shall see that Icicle has no need to scan the database and that the selection process is trivial, all matching rules work in parallel.

The study on non-programmers revealed that non-programmers more commonly described program actions in terms of a user of a program than in terms of a programmer. In Icicle the fact that programs are produced as they are run can be seen as matching this expectation – the user is switching between the roles of player and programmer.

Interactions as an understandable conceptual model

Guideline 7 concerning explicit models is related to Norman's conceptual models (Norman, 1983). A conceptual model lies between the actual model of the physical (or external) system and the mental model of a person using the system. A good mental model makes working with the system easier and decreases the likelihood of mistakes. A good conceptual model makes it easy to develop a good mental model. Conceptual models can be taught and this has been shown to improve performance in some situations (Yeshno & Ben-Ari, 2001).

As many interactions occur in an Icicle program, a user's understanding of a program could quickly become chaotic as multiple things happen simultaneously. Because of this, the model explicitly presented to users must be easy to grasp and remember. The Icicle model is simply, "The computer is constantly watching all of the performers' and users' actions and changes a performer's behaviour when an interaction occurs."

Interaction computing and variables

The performers in Icicle, as objects that are displayed on the screen and move around the world, require state representing their positions, their sizes and their orientations. These values are stored in

variables. As Icicle is an open-ended programming environment, these values have to be accessible in some way to users.

In keeping with guideline 2, the variables are visible and directly modifiable by the user. In keeping with guideline 5, they change immediately as the performer changes in the world. If the user moves, rotates or resizes a performer the variable values are updated immediately. Conversely, if the user types a value into a field representing one of the variables, the corresponding change is made to the performer in the world.

In order to produce more sophisticated programs and computations, the user needs to be able to create new variables. This produces a problem. Variables have to be named. As we saw earlier, Icicle is designed to avoid a reliance on text in most situations, so that children still learning to read can use it. Since variables are the most abstract entity in Icicle I assumed that they would be hard to understand, used least often, and only used by older children. These assumptions turned out to be correct (see Chapter 7). Because of this, it made sense to allow textual names for the variables.

Doing things with variables

Interactions on variables were one of the earliest design decisions for Icicle. *Interaction computing* was only used as a theme in the design of Icicle because it could be generalised to the fundamental containers of state in programming languages – variables. This was done by viewing variable assignment as a form of collision. When a variable is assigned a value, there is a search to find a rule associated with the variable that matches the value being assigned. If such a rule is found the instructions it contains are executed, if no such rule is found the user is asked to supply one.

Performers as objects

Icicle activates rules when the state of a performer in the world changes. If the state change does not match an existing rule, the user is asked to describe what to do in this case. Other programming systems for novices have used the same idea, in particular "Klik & Play" and "The Games Factory" game producing environments (Clickteam, 1994) and their derivatives. These programs maintain global state and instruction tables. They represent a game as a large table of all possible events that can occur in the game and the corresponding instructions to execute when an event occurs, as in Figure 27.

The rules displayed in this table can be filtered to show only those rules caused by particular objects or that affect particular objects, but there is no conceptual grouping of rules with objects. It is easy to understand why. Imagine a rule associated with the collision of two objects. When the objects collide, we want one of the objects to disappear, the other object to change its direction, and a score object to be incremented. With a global rule table the rule is represented by a line such as "when object *a* collides with object *b*, delete object *a*, change the direction of object *b*, and add 1 to the *score* object". This keeps all instructions associated with the collision together. The instructions are encapsulated within one rule.

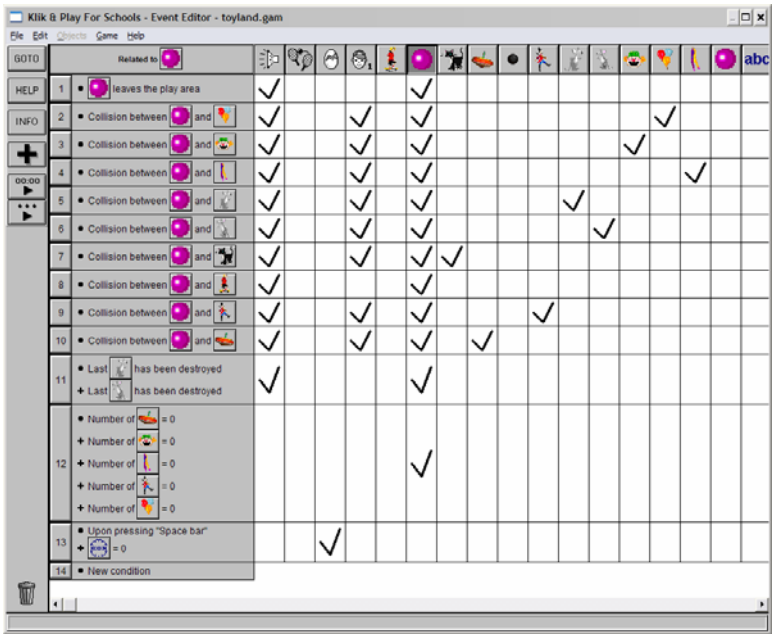


Figure 27 – Event rules and instructions in Klik & Play.

The main reason Icicle does not take this approach is the desire to make the performers real things, with their own behaviours. With a global rule table, the instructions are actions that happen to objects, not the reactions of objects to changes in the world of the program. The Icicle approach is more of an object-oriented view of the performers.

In a study on non-programmers (Pane et al., 2001) it was seen that the participants attributed state and behaviour to program entities but no corresponding understanding of inheritance and polymorphism was observed. This means that in the taxonomy of Cardelli and Wegner (Cardelli & Wegner, 1985), the non-programmers saw programming as dealing with an object-based environment. (An object-based environment is similar to an object-oriented one but without the inheritance.)

In Icicle, the objects are performers; they have appearances, behaviours and internal states, but there is no inheritance of one type of performer from another type. In this sense, Icicle is an object-based environment.

Prototype-based performers

Icicle lies somewhere between a normal object-based environment where classes are constructed and objects are made from the classes, and a prototype-based object-oriented system (Ungar & Smith, 1987) where objects are constructed from other objects used as templates. In Icicle, very little distinction is made between a class and an object. From the point of view of a computer scientist, a class is represented by a performer box and a performer in the world is an object. For users this distinction is not so obvious. When a user wants to put a new type of performer in the world, he or she clicks on the "create new performer" button, as we will see in Chapter 5. This creates a new performer box. The name of the button was chosen deliberately. Even though the user is actually creating a new class, the intention of the user is to create a performer to drop into the world.

Dropping a new performer into the world will immediately cause Icicle to request a rule to be defined. Whenever a rule is being defined at the request of Icicle, one particular performer will have been the receiver of the event and Icicle will have selected that performer in the world so that the user can produce the instructions for the rule, using that performer as an example. Even when several performers receive an event simultaneously, Icicle will select one of them for the purpose of requesting the rule instructions. For this rule, the selected performer is acting as the prototype for all other performers of the same type. In this way, Icicle is a prototype-based system. Rather than one prototype performer, as in a standard prototype-based system, any performer can be a prototype of its class by virtue of being the performer used to define a rule. Another difference from standard prototype systems is that the performer objects are not created by copying another performer. The user cannot create a performer, modify some aspect of it and then copy the modified performer. We have already seen that modifying the rules or appearance of any performer instantly modifies the behaviour and looks of all performers of that type. All existing performers would also take on the modified characteristics.

Icicle is thus an object-based programming environment with a form of prototyping.

CLOSENESS OF MAPPING

In his influential chapter on cognitive engineering, Norman (Norman, 1986) introduced the twin Gulf concepts: the Gulf of Execution and the Gulf of Evaluation. The Gulf of Execution is the gulf between the intention a user of a system has and the actual physical processes the user has to perform in order to complete the task that satisfies the intention. The Gulf of Evaluation is the gulf between the user's perception of the state of the system after the task has been completed and the user interpreting that information and comparing it to the original intention.

Closeness of mapping is a measure of how closely a notation represents the tasks it was designed to perform. For a notation representing a program, this dimension has two aspects corresponding to the two gulfs: the directions or commands that must be given to the system to produce the program, and how the program is displayed when it is viewed by the user.

One way the gulfs can be reduced is by moving the system closer to the user. In other words, for the Gulf of Execution, making the actions necessary to achieve the task close to the intentions of the user. Here programming by demonstration (PBD) is useful to reduce the gulf between the intention of producing a program and the production of the program. If the user knows how to carry out some task, he or she can use that same knowledge to produce a program that automates the task. Direct manipulation (Shneiderman, 1983) can further reduce the gulf by moving the actions required to carry out the demonstration closer to the intentions. For example, if the user intends rain drops to fall from a cloud, dragging a raindrop from a cloud is an action that is close to the intention. Icicle uses both direct manipulation and a simple form of PBD. Both of these techniques attempt to reduce the cognitive processing that must take place to convert the original intention into actions.

As Icicle is a programming environment, there are two ways to consider the Gulf of Evaluation. The evaluation can be at the level of, "the program does what was intended". The performers either

move and change as the user intended or they do not. The same applies to the values of variables, which can be shown on the screen – they change as their values change. Alternatively, the evaluation can be at the level of, "I want to see how the program works". This is important in a variety of situations. Someone else may have created the program and the user wants to understand how it works. Similarly, the user may have created the program but has forgotten exactly how it works. In the case of programs produced by demonstration, the user did something that has been converted into a programming language or some internal representation and the user wants to inspect the representation.

Providing a listing of the generated program that is useful to the user is one of the central problems in PBD systems (Myers & McDaniel, 2001). Supplying a listing of a conventional textual program is not acceptable for novice users, in this case children. The connection between a textual program listing and the actions of objects on the screen is not clear enough. There is a gap in the novice user's understanding between the textual representation and the effect produced. Repenning and Perrone refer to this as the "PBE representation chasm" (Repenning & Perrone, 2001). Icicle deals with this chasm by animating instructions.

Demonstrating instructions

Icicle programs consist of "when" rules. When something happens, Icicle finds a rule with the new interaction as its condition and begins executing the instructions in the result part of the rule. If there is no rule for the "when" condition, the system stops and asks the user what he or she wants to happen in response to this situation. The user must then demonstrate the required behaviour to the system. In keeping with guideline 6, Icicle provides two different ways to convey program instructions. There are a number of instructions that can be demonstrated by manipulating performers in the world, and all instructions can also be selected explicitly by clicking on instruction buttons. These are similar to the control techniques used by ICE in Chapter 2. Exactly what can be demonstrated, and how, is described in Chapter 5.

Representing instructions with animation

The other side of the closeness of mapping coin is how well the system bridges the gulf of evaluation. In the case of a programming environment, how easy is it for the user to determine that the system is representing the required program? How easy is it for the user to understand a program written by someone else, by inspecting its representation?

In ICE, subroutines were represented clearly, as a reduced image of the picture produced by the instructions. Unfortunately, the actual turns and moves were only indicated with static icons showing a turn or a move and a textual tooltip that described the actual instruction when the pointer hovered over the instruction button. This made understanding the sequence of instructions difficult and needed to be improved in Icicle.

The Icicle system uses animations of the instructions as the program listing. There is a true closeness of mapping between the program code and the desired behaviour; they are virtually identical. The move instructions show animations of the performers moving through the correct

number of relative pixels, the turn instructions have animations of performers turning in the correct direction through the number of degrees specified by the instruction, and the morph instructions have animations of performers changing from one shape to another. In Chapter 7, we shall see that the animation of the instructions led to a significant improvement in the ability of children to interpret instructions.

Even though Icicle was designed to be usable by children who are not confident readers, it still makes sense to include text where it is helpful. Therefore, all instructions also include brief textual descriptions beneath the animations, e.g., "move 147" or "turn -34". These can be used by children who can read and want details such as the distance moved.

There are several problems with animating instructions. One is that the shape used in a rule to represent the performer can change during the list of instructions because one of the instructions is a morph. In order to maintain the closeness of mapping between the instructions and their results, the new shape must be used in instructions that follow the morph. This has to work even when the user edits rules by moving instructions around and deleting and inserting instructions. Thus, an animation showing a turn will use a different shape for the performer if the instruction is moved from its original position, before a morph instruction, to a new position, following the morph instruction. The performers in the animations have the shapes they will have when the instructions are executed.

The second problem is to do with screen area or real estate. Some of the instructions can cause results that require animations of arbitrary sizes. How should an instruction like "move 500" be handled? If a full-size animation is shown, it takes up a very large amount of space to represent one instruction. The solution chosen was to shrink the animations so that they fit into a maximum sized window. Small moves (and zooms) do not require any scaling but larger ones do. This scaling damages the aim of closeness of mapping. It was unknown how children would react to this scaling, and whether it would cause difficulty in understanding the instructions.

In order to convey the concept of instructions being performed in sequence, Icicle animates the instructions in a rule one after another, and highlights the instruction currently being animated. Children can easily see what happens when the rule executes and in what order.

The conditions that cause the rules to activate are also animated. Even for children unable to read the descriptions beneath each animation, it is clear what is being represented here. Parallel instructions (see *Ease of Parallelism*), also provide a high closeness of mapping between the instructions and the actions they represent.

Animations as distraction

The animations in Icicle convey information directly related to the task at hand, unlike animations that are used merely for decoration. Even animations included only for aesthetic purposes do not appear to distract learners to the extent of causing a detrimental effect on the learning (Rieber, 1996). If this is generally true, it would mean that Icicle could animate all buttons and rules continuously without regard to any distractive consequences, but in this case the animations would lose the ability to direct the user's attention to relevant items of interest and so it was decided to activate animations only when the user's attention, as identified by the position of the pointer, was on a rule or a

command button. The command buttons only animate when the pointer moves over them. If a list of several rules is visible on the screen the instructions in each rule only animate if the pointer is somewhere over the box containing the rule.

EASE OF PARALLELISM

The real world has very few constraints on parallelism. It is common to have several events occurring simultaneously on the same object or within the same environment. To appear realistic any simulation of the real world must deal with parallel events in a way that approximates the reality being modelled. Most game programs must also cope with simultaneous events. There may be several objects represented within the game that carry out their own behaviours in parallel. Interactions can occur between the objects, which may cause behaviours to change or further behaviours to be added to those already occurring. Maintaining and controlling parallel behaviours is essential to the realism discussed under *lively concrete objects*.

In operating systems and within computer architectures we distinguish many levels of parallelism depending on the choice of computational unit. Most operating systems allow parallelism at the process level or the process itself can be divided into subsidiary computational units usually referred to as threads (Bacon & Harris, 2003). Specialised computers are usually required in order to use parallel computational units smaller than this efficiently e.g., array processors (Duff, 1980) or dataflow architectures (Gostelow & Thomas, 1980).

We know that there are many difficulties with parallel programming. Operating systems textbooks (Bacon & Harris, 2003; Silberschatz & Galvin, 1998) include examples of race conditions where the results of programs cannot be predetermined due to timing differences. The standard solution to this problem is the creation of some sort of control mechanism that enforces sequential access to shared objects. Fortunately, this can be done fast enough to maintain the illusion of parallelism. Unfortunately, this solution is prone to producing deadlocks. Programming with parallelism requires careful thought and debugging such programs can be exceedingly difficult. As parallelism is required for many programs, we must include the capability in our programming languages and environments. How can such parallel behaviour be included in a programming environment intended for children or other beginning programmers?

Parallelism in other environments

Of the various programming environments used by children, ToonTalk, Stagecast Creator, AgentSheets and NetLogo (Wilensky, 1999) provide three different solutions to the parallelism problem.

ToonTalk is designed entirely around parallelism. Robots represent methods (or clauses) and each robot does only one thing at a time. In fact, in a team of robots only one is ever working at a time. One of the most powerful things a robot can do is to create a new process and as long as the creating robot does not require a value from the new process both can continue in parallel. Processes are very lightweight in ToonTalk and many thousands can be produced without negative side effects such as running out of stack space. Traditional function calls are simulated by one robot producing a new

computation and another robot waiting for the results of that computation. As requests to robots are queued until they can be handled and since each robot does only one thing at a time and there is no shared state between robots, most of the problems of parallelism are avoided.

Creator provides parallelism because many matches can occur during the same time period or clock tick. Even though many rules can be run in the same clock tick, each rule is carried out to completion before the next rule runs. The characters are checked for active rules in the order they were added to the stage. The rules for each character are inspected in turn and the first rule with a condition that matches the world is executed. There is also a type of rule precondition, "do all and continue" that allows several rules to execute (one after another) in the same clock tick for the same character.

AgentSheets works in a similar way to Creator, with rules collected together into groups called methods. Only the first rule with a satisfied condition is executed from all of the rules in the method. Parallel activities for individual agents are provided by multiple methods being active simultaneously. It is interesting to note that the lack of true parallelism has been commented on by some users of AgentSheets.

> *The fact that a single rule is executed in a method at each execution cycle, and not all the rules whose conditions are satisfied, is counter intuitive in many occasions. Most of my students' difficulties in modelling behaviours have to do with this.*
>
> (Carvalho, 2000)

NetLogo, based on StarLogo (Resnick, 1994), provides massive parallelism by extending Logo and controlling many turtles and patches (stationary screen areas) in parallel. In this text-based language, not only can all of the turtles and patches carry out instructions simultaneously, but also each turtle or patch can work on more than one block of code at a time.

I refer to a sequence of one or more instructions carrying out some particular task as an instruction stream. Each of the environments described here has its own way of defining instruction streams. In ToonTalk, it is the instructions associated with a robot. In NetLogo, it is the instructions in a block of code. In Creator, AgentSheets and Icicle, it is the instructions derived from a rule.

Icicle allows arbitrary parallelism at multiple levels. It not only works at or beyond the level of the other systems described here, where each performer can be executing the instruction streams from many rules at the same time, but it also does something unique – within a single instruction stream several instructions can be executing in parallel. This information is summarised in Table 2. (This table is referred to in Chapter 6, when discussing the implementation of scheduling.)

Environment	Instruction stream	Computational object	Multiple streams per object (on by default)	Interleaving of instructions from different streams	Parallel instructions within a stream
ToonTalk	Robot instructions	robot	No	Sometimes	No
Creator	Rule instructions	character	Yes (no)	No	No
AgentSheets	Rule instructions	agent	Yes (no)	No	No
NetLogo	Block of code	turtles and patches	Yes (yes)	Sometimes	No
Icicle	Rule instructions	performers	Yes (yes)	Yes	Yes

Table 2 – A comparison of parallelism in the different environments.

It is reasonable to want the performers to do more than one thing at a time. If we have a performer representing a dog running we want the object to change its shape as it moves across the screen (Figure 29), otherwise it looks as though it is sliding over ice (Figure 28).

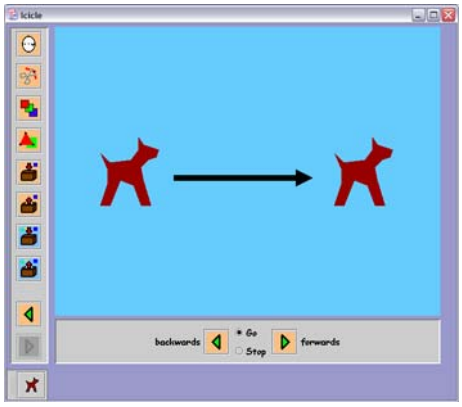


Figure 28 – A dog sliding across the screen in Icicle.

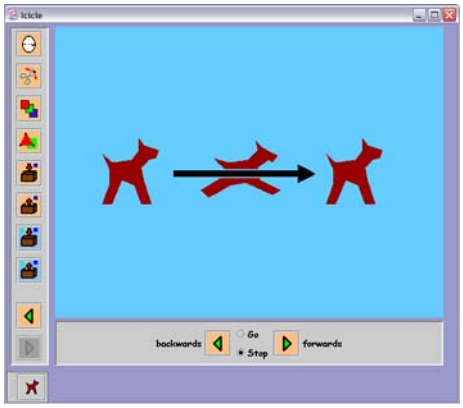


Figure 29 – A dog running across the screen in Icicle.

How Icicle makes the production of such parallelism simple is described in Chapter 5. To maintain the closeness of mapping between the parallel instructions and their results, the animations representing the instructions show the combined behaviour of all the participating instructions.

The implementations of parallelism in Icicle and the four other environments are described in Chapter 6.

FLEXIBILITY

People have different styles when it comes to solving problems. Some people carefully decompose problems and completely plan the solutions before attempting to implement them. Others seem to "play around" with problems or in Papert's words, "Use what you've got, improvise, make do". Turkle and Papert refer to this approach as *bricolage* (Turkle & Papert, 1991). Sometimes the problems themselves are not well defined. Sometimes playing around with a shape or an idea leads to a discovery that is interesting and worth pursuing. It may just be fun, and with children, that is reason enough in itself.

It is easy to criticise both of these extreme problem-solving styles. In some situations, it is impossible to solve a problem using a strict top-down approach, in others it is impossible to solve problems by playing around. They both have their place and everyone uses both of them at different times. Any computing environment, not just those for children, should make it easy to employ both methods of control.

The bricolage style of playing around places far more demands on a programming environment than a nicely structured approach. In order to fiddle with the program it needs to be changeable at any time and in any way. Restricting ourselves to a traditional design-write-compile-execute-debug cycle severely limits the ease of making changes. The *lively concrete objects* techniques discussed above are the way Icicle allows changes to happen throughout the development of programs, even when programs are running. Many other programming environments, including spreadsheet programs, support bricolage in other ways.

Bricolage with Icicle

Several interrelated components affect the outcome of a program. In Icicle, they will all be associated with the performers. A child may want a performer to change shape when some event occurs. It is highly likely that the child will not have constructed the new shape before the time it is needed. Thus, shapes must be able to be constructed and modified at any time during the execution of a program. Even more dramatically, a different type of performer may be required in response to an event. Behaviours of performers will need modifying as changes are made to their appearances and as they interact with other performers in the environment.

Therefore, in Icicle there are no limits on what can be changed or when performers, shapes or behaviours can be produced. Any changes are immediately apparent, e.g., when a performer's shape is altered all performers showing the same shape have their shapes automatically updated.

It can be argued that there is a danger of producing unstructured, undisciplined, and muddled programs using such a flexible environment. Of course, this can be done, but this can be done even with environments that are more traditional. By providing a flexible environment that *bricoleurs* are comfortable with, we are not depriving more methodical design-first types from using the system. The system may not enforce design-first, but there are no impediments that make it inconvenient to use it this way. When we look at some programs produced by children in Chapter 7 we shall see that the pay-off, of letting the environment provoke ideas by allowing the flexible approach, is worth it.

Several of the Cognitive Dimensions of Notations are associated with this type of flexibility. *Premature commitment* is when an environment forces choices to be made in a particular order. An environment that requires all shapes to be finalised before a performer can be added to the world would suffer from premature commitment. As we have seen in Icicle, shapes and behaviours can be changed at any time.

Another of the dimensions is *provisionality*. Provisionality is concerned with how easy it is to explore. If something is done to try out an idea, can it be undone easily if the result is not desired.

Icicle employs comprehensive undo facilities; not only when constructing shapes and demonstrating rules. Running Icicle programs can themselves be undone and redone with modifications.

Progressive evaluation is the dimension concerned with being able to assess the state of a solution partway through its construction. In one sense, there is no such thing as an incomplete Icicle program. There are no syntax errors that prevent programs from being tested if they are only half-finished. Programs can not only be tested at any time, but the usual way to write an Icicle program is by running it and adding bits to it as the program progresses.

THE ICICLE INSTRUCTIONS AND INTERACTIONS

Instructions

How were the instructions for Icicle chosen? Guidelines 1, 10 and 11 give some assistance in determining possible instructions. An attempt was also made to keep the number of instructions to a minimum and yet still supply the ability to produce the effects needed in games, simulations and other programs.

Move

Some instructions were easy to choose. The Icicle performers were designed to move around the world so there had to be instructions that enabled this. The *move* instruction took over the job of Logo's (and ICE's) FORWARD and BACK instructions. The BACK instruction is useful in Logo to remove the necessity of negative numbers. Negative numbers are seldom taught to children before 10 years of age. (In user testing I found that negative numbers were easy to understand by 9 and 10-year-olds, at least in the limited sense of meaning "go the other way".)

It was observed in ICE that children seldom used the move backwards command since a normal drag of the turtle produced both a turn and a move instruction. Icicle works the same way.

Turn

Because most turns are demonstrated with drags, there is only the one *turn* instruction in Icicle. A clockwise turn is given a negative value in the instruction. As with the move instruction this is not a barrier to understanding the instruction when it appears in a rule since it is represented by an animation showing, not only the direction, but the amount of the turn.

Morph

A *morph* instruction was required in order to change a performer's appearance from one shape to another.

Zoom

A *zoom* instruction to increase or decrease the size of performers was added after very early user testing. The original implementation based the size of performers on the sizes drawn in the shape editor (see Chapter 5). Children working with the early version found it too difficult to create performer shapes of the size they wanted. They kept producing performers that were not in scale with each other. By allowing performers to be resized at anytime this was no longer a difficulty. Adding a resize instruction was the programmable counterpart to this.

Make a sound

Because of guideline 10, it was always the intention that Icicle would have a way to produce sounds. However, the version first used in the user study did not have this ability. The outcry from children using that version meant that a *play* sound instruction was added as soon as possible. In keeping with the closeness of mapping theme, when the sound instruction is animated it causes the actual sound to be played.

Creation and deletion

Performers need to be created and deleted as Icicle programs run. This is the case for situations such as a space invaders game where spaceships are produced as the game runs and where they are deleted when hit by a missile. The same is true in a simple text editor program; typing produces performers representing the letters typed and pressing the delete key deletes them.

Performer deletion removes a performer from the world and stops it taking any more part in the program. As described under *lively concrete objects* all performers are their own masters – one performer cannot directly manipulate another performer. Because of this, performers may only delete themselves. This is more natural than it might appear at first sight. Imagine a collision between a missile and a spaceship. When the interaction occurs, both the missile and the spaceship will be notified, and if the spaceship is to explode and disappear, it does this with a morph to an explosion shape, followed by the delete instruction. It is not the missile that has caused the destruction; it is the collision between the missile and the spaceship.

Bring to top

Performers move around the world and sometimes overlap each other. In this case, it may be important for one performer to appear on top of the other performer. Thus a *bring to top* instruction, which brings a performer to the top layer in the Icicle world, is needed. All performers can be thought of as existing in different layers of the world. The most recently added performer is in the top layer and partially obscures all other performers that occupy the same area of the world.

Set a variable

As performers have variables, there must be some way to assign a value to the variables. The *set a variable* instruction causes a rule to execute. Expressions can be evaluated and assigned to variables. How this is done is covered in Chapter 5.

Tell a performer

Icicle is a programming environment where computations are carried out by the performers in the world. Because of this, action at a distance is required to allow a performer or an interaction on one side of the world to cause some effect on the other side of the world e.g., a collision between two performers might be the necessary condition for the score variable of a third performer to increment.

As we saw, Creator deals with this problem by allowing a rule associated with one character to change the properties or variables of another character. In order to pass information between characters that are not related by a rule condition, Creator uses global variables.

Both of these solutions were wrong for Icicle. In the first case, one character modifies the state of another character. In Icicle, each performer is in control of its own state. One performer cannot reach in to another performer and change its properties.

The second case, using global variables, had a similar difficulty. In Icicle, the global variables could have been associated with the world. The problem, ignoring the distaste computer scientists have traditionally held for global variables, was that Icicle performers only do things when interactions occur. In this case, some performers would require instructions that are executed when a value is assigned to a global variable. This means that performers would be changed by state outside themselves, and would have to be able to modify that state directly.

A message passing system avoids these external considerations. AgentSheets uses two forms of message passing to solve the same problems, a broadcast form and a specific form that sends a message to an agent in one of the surrounding grid positions.

There were two design problems encountered when trying to incorporate a message passing scheme in Icicle. The first was the need to give messages names. There has to be some way of identifying messages so that when a performer receives a message it knows what message has been sent to it. Different messages should cause the performer to do different things.

Because Icicle is designed to be usable by children with limited reading and writing skills, the names for messages are graphical. Also, rather than expecting the children to make their own names for messages the system allocates names automatically. The form of these messages and how they are presented to the users are described in Chapter 5.

The other design problem encountered when adding the message passing system was concerned with directing the messages. How does the user specify which performer should receive the message? When demonstrating a rule for the first time it is possible to select a particular performer in the world to receive the message. Unfortunately, this solution has many problems. The interaction that causes the message to be sent as the program runs, might occur at a time when the original receiver is no longer in the world or it may have been replaced by another performer of the same type that the user now wishes to receive the message. This also raises the question of how will the particular performer be identified in the message sending instruction? Another difficulty with this solution is how to edit the instruction at a later time or even create the instruction initially if the intended receiver is not currently in the world.

Sometimes a performer will have to send messages to all performers of a particular type – a broadcast message. This can also be used to send messages to a single performer if there is only one performer of that type in the world. It turns out that this solves the problems above. There is no *original receiver* – any performer of the required type will get the message. What if the user wants to send a message to a particular performer? In almost all situations where the user wants to send a message to a single performer it is easy to make sure that the performer is the only one of its type in the world. On the occasions this is not the case, all receiving performers can check to see if the message was intended for them, before acting on it.

Turning towards another performer

Many games require one performer to move towards or face towards another performer. Rather than providing trigonometric functions and having to use the position variables of performers, I supplied a *turn towards* instruction in accordance with guideline 11.

Interactions

Apart from the instructions, the other aspect of an Icicle rule is the interaction that causes the rule to execute. Some interactions will be caused by the movements of performers in the world, some by the user directly controlling a performer, some by changes to variables, some by communication between performers, and some by special situations.

Changes in the positional relationships between performers in a world can be used as interactions. There are many possible changes of this type, e.g., a performer moves from being on the left of another performer to being on its right, a performer collides with another performer, a performer that was partially overlapping another performer moves to be completely enclosed by the other performer. Of all of the possible interactions of this type, the collision interaction is the most obviously useful. Many games and simulations require things to happen when performers collide. In order to keep the number of interactions to a minimum Icicle only uses collisions and their opposite, "no longer touching", as positional interactions. Many other positional interactions can be dealt with using variables.

There is no need for Icicle to provide mouse control interactions because performers can be manipulated within a world as the program runs, e.g., the user can grab a rectangle performer and drag it around the world using it like a bat. Key presses are another matter. Many games can be controlled by keys pressed by the user; therefore, Icicle provides key press interactions.

Interaction computing includes the ability to execute a rule when a variable is assigned a particular value. Rather than having as many rules for each variable as there are values the variable can accept, Icicle can set ranges for variable interactions, for example, when this variable is assigned a value greater than 100, execute this rule.

There is an instruction to send a message to a performer. This means that Icicle requires an interaction to execute a rule when the performer receives the message.

There must be special interactions to initialise programs, e.g., when an Icicle world is loaded the performers may need to be initialised. Similarly, when a performer is created it may need to be initialised.

Lastly, performers frequently have to produce behaviours when they are not involved in interactions, e.g., a car keeps moving along a road until it collides with another car. To satisfy this requirement Icicle provides a default rule type that is executed when nothing else is happening to the performer.

COMPLETENESS OF INSTRUCTION SET

These instructions and their associated interactions were decided upon by considering scenarios of performers interacting as objects do in games and simulations. As we shall see in Chapter 7 they facilitate a wide range of program types. Is it possible to be more formal than this?

Some preliminary work has been done on design patterns for visual languages producing games or simulations, i.e., what types of situations commonly occur in these types of programs and how can they be produced. Lewis et al. have produced some patterns by looking at proposed simulation scenarios demonstrating something of community concern such as school yard violence (Lewis, Rosson, Carroll, & Seals, 2002).

There are four patterns described in this work: *You're in my space, I'm in front of you, I don't care,* and *We regulate*. Rather than specifying individual instructions, they describe relationships and when changes in relationships should cause changes in behaviour.

You're in my space

The first pattern is a spatial conflict resolution pattern, where the presence of one object modifies the behaviour of another object occupying the same space. This is achieved in Icicle using the "collides with" interaction. The rules associated with the interaction can modify both performers and it is possible for one performer to use values from the other performer in its response.

I'm in front of you

The next pattern deals with objects occupying the same space and choosing which one is on top. In Icicle, this is a subproblem of the previous one. Occupying the same space is signalled by a collision interaction and one of the performers can use the "bring to top" instruction to appear on top.

I don't care

I don't care is applicable when an object is supposed to carry on with some behaviour regardless of the actions of other objects and their positions and actions in the world. In Icicle, the interactions caused by other performers, such as collisions and messages, can be ignored. Each performer is in control of its own state and reactions.

We regulate

We regulate is concerned with global control where the behaviour of an object is modified by some overall change in the simulation. In traditional systems, this would be done using a global variable. In Icicle, it depends on what is causing the global change. If the change is caused by the user typing a key the "key pressed" interaction is sent to all performers that have a rule for that particular key. If the change is caused by something else, a performer in the world must broadcast a message to all performers that are supposed to react to the change.

There is nothing to stop Icicle performers polling a value in another performer in order to see if their behaviours should change, except that in Icicle the idiom is reversed; the performer with the

value that needs checking would repeatedly send a message to all concerned performers and they would check the value when they received the message.

SETS OF RULES

These patterns do illustrate a difficulty in Icicle. How can one interaction cause two different behaviours, depending on other changes in state? This can be accommodated by using a variable as a flag. When the interaction occurs, the performer can check the flag and do different things depending on its value. In an early draft of the Icicle design, performers were going to have sets of rules, only one of which was active at any time. There was to be a way of flipping between sets of rules as circumstances changed. This is a nicer solution but was regarded as a violation of guideline 4 because the sequence of instructions to be followed would depend on some hidden state.

THINGS LOST FROM ICE

Although Icicle is built on the foundation of lessons from ICE, several things did not come across in the design. In particular, the automatic loop recognition and the easy introduction to procedures have been lost.

The loop recognition should be introduced in a subsequent version of Icicle. Currently, loops are provided by the default interaction and by repeatedly sending messages and assigning values to variables (see Chapter 6). In the cases where the user wants to repeat a sequence of actions a certain number of times, the loop recognizer, combined with an editor to alter the number of repetitions, would be much simpler than having to decrement a variable and repeat the actions until the variable reaches zero.

In ICE, a number of instructions could be turned into a procedure by dragging a red box around them. The same thing could be done in Icicle but there does not seem to be as much need. In ICE, the program was one long sequence of instructions, with some loops and some procedure calls in the sequence. In Icicle, the program is a large number of rules caused by interactions. The rules themselves seldom consist of many instructions. On the other hand, procedures can be simulated in Icicle by a performer sending a message to itself. These procedures are run in parallel rather than in sequence with the sending rule.

THE USER-LEVEL DESCRIPTION OF ICICLE

This chapter describes how the design from Chapter 4 is represented to the user. The user interface is shown as well as the actions a user makes when producing a program in Icicle.

CONSTRUCTION OF ICICLE PERFORMERS

The first thing a child does when producing an Icicle program is to construct a performer, or more accurately a performer template. Figure 30 shows the initial appearance of the Icicle program, with a number of tools down the left-hand side, a large central area where the programs execute, called the world, and a control panel underneath the world. This is very similar to the layout of Creator (Figure 31). By clicking on the "create a new performer button" a performer box appears and the child has to choose the base shape for the new performer (Figure 32).

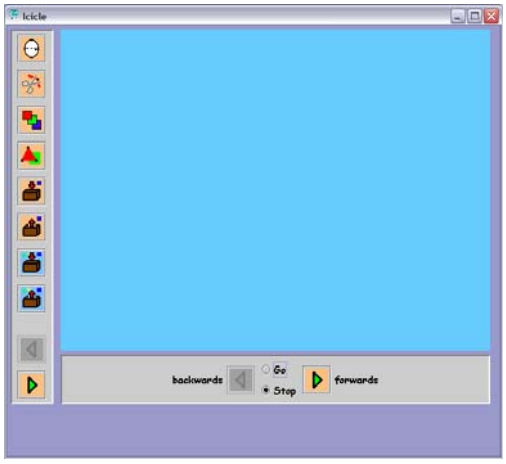


Figure 30 – The Icicle world and controls.

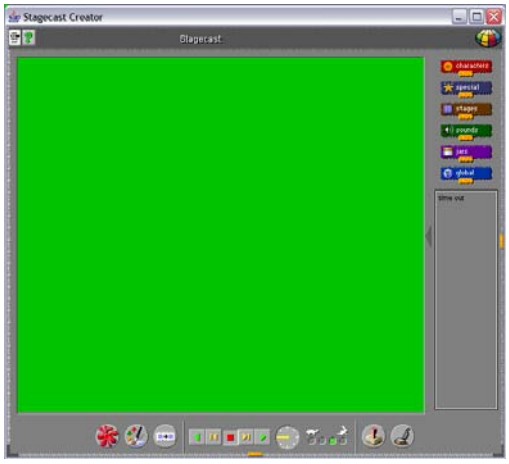


Figure 31 – The Creator stage and controls.

All performer shapes are polygons. As we shall see in Chapter 7 children requested that a circle shape be added to the options. I mistakenly thought the octagon would suffice.

When one of the base shapes is clicked on, the performer box changes to show the selected shape in a shape editor panel called "looks" (Figure 33), as in the different *looks* of the performer. This panel allows editing of the shape in the most common ways, without requiring the user to select tools to perform the operations. In particular, points can be added to a shape by clicking near one of the lines of the shape. As the user moves the mouse pointer over the window, the pointer image changes from an arrow to a hand when the pointer comes close to the lines that constitute the shape. When the principal button on the mouse is pressed, a new point is inserted in the shape at that point and the corresponding edge is replaced with two edges leading to the new point. Adding a point does not require any tool selection – in a similar way to the so-called post-WIMP interfaces (Beaudouin-Lafon et al., 2001). Likewise, moving a point is done by dragging points without selecting a tool.

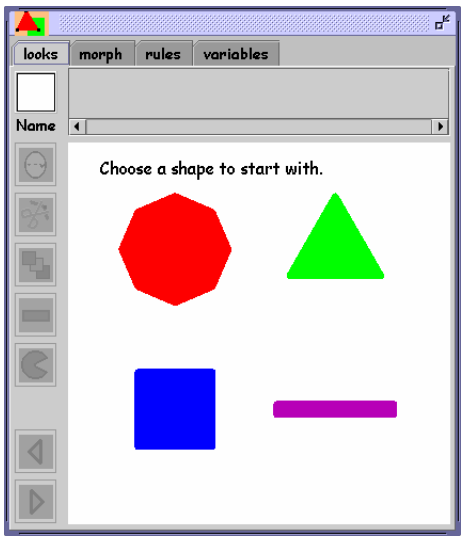


Figure 32 – A fresh performer box.

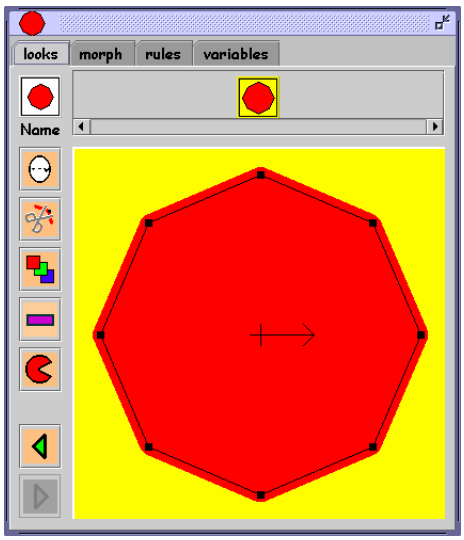


Figure 33 – A shape for this performer.

Points are deleted by selecting the scissors tool and then clicking on the points to be deleted. The same icon is used throughout Icicle to represent the deletion of objects. The screen pointer changes to a pair of scissors when the user is in delete mode and as the scissors move close to a point, the point is highlighted with an orange circle, indicating that the point can be deleted. The delete mode is deactivated by either clicking on the scissors tool again or by clicking somewhere in the window away from a point that could be deleted. This means that multiple points can be deleted without having to reselect the scissors tool. If the shape is reduced to three points, the delete mode is deactivated and the scissors tool becomes inoperable. This is because shapes are always polygons.

All performer shapes have a facing direction. When a performer in an Icicle program moves it will normally move in the direction the shape is facing. When a shape is being edited, the facing direction is indicated by the arrow in the centre of the window pointing to the right (see Figure 33). If a child starts drawing a shape facing upwards, such as a rocket, it can be rotated so that it is pointing in the facing direction, by clicking on a rotate tool button. Similarly, there is a tool to reflect shapes in the vertical axis.

All tool buttons in Icicle animate as the pointer moves over them, demonstrating the action they represent. This was another choice following the guideline of providing a lively interactive interface. Tooltips describe the actions of the buttons for children who can read.

In case the child makes a mistake when changing a shape, all operations in the shape editor (and in most other situations in Icicle) can be undone and redone. In Chapter 6 we will come across some interesting questions with respect to undoing and redoing operations but in the shape editor, it is straightforward.

Extra shapes can be added in the "looks" panel and they appear across the top of the editing area in a shape list (Figure 34). There is nothing that forces children to base extra shapes on the shapes they have already created. One way to add a new shape to a performer is to call up the four starting shapes' window again, but there is a simple way to base extra shapes on an existing shape – dragging the small representation of a shape in the shape list and dropping it into open space in the list makes a copy of that shape. The copy is then available for editing. Whenever a performer shape is dragged, the

pointer changes to the shape and colour of the shape being dragged. Only one shape from each performer box can be edited at a time. The shape currently being edited not only appears in the editor window but its representation in the shape list is highlighted with a yellow background. The shape currently being edited can be changed either by clicking on another shape in the shapes list or by dragging a shape from the shape list onto the editor window.

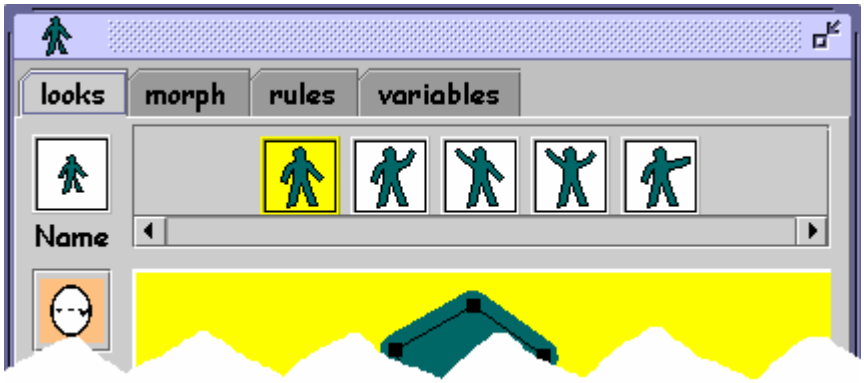


Figure 34 – Multiple shapes for this performer.

Name shape

The shape over the word "Name" in the performer box (Figure 34) is known as the "name shape" and is used to represent the performer in several situations: within rules, in the title bar of its performer box and on its minimized performer box. By default, this is the first shape created for the performer but it can be changed by dragging another shape from the shape list and dropping it on the square above the word "Name".

Morph tester

Before the child uses a morph from one shape to another in a program, he or she can experiment with morphing in the performer boxes. The performer must have at least two shapes in the shape list. The morph panel (Figure 35) shows the shape list above the area that will demonstrate the morph. On the left-hand side of this panel, there is a vertical slider control and boxes for three shapes: one at either end of the slider and one in the top left-hand corner. The child drags a shape into the box directly above the slider and another shape into the box below the slider. Then, by moving the slider up and down, the child can see the morph between the two shapes. The result is shown in large in the main area of the panel and in small, the default size of a performer in a world, in the box in the top left-hand corner. In Figure 35 the conductor shape is approximately two thirds of the way from having its arms down to having them both raised.

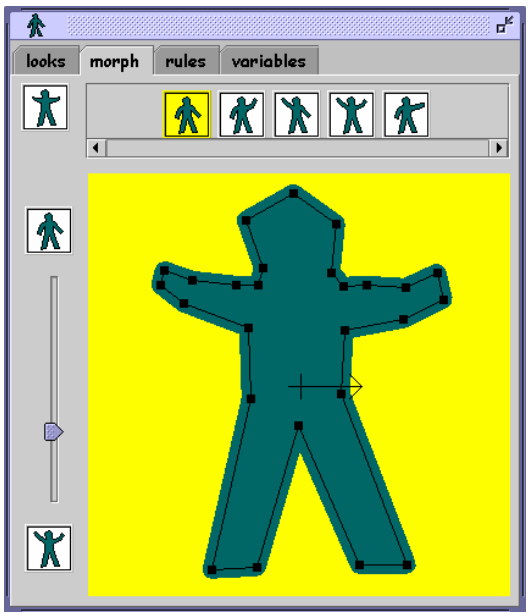


Figure 35 – The morph panel in action.

ADDING PERFORMERS TO THE WORLD

After creating one or more performer shapes, the child can add performers to the world. Icicle allows new performers to be dragged to any position in the world from the performer boxes. If the box is minimised (Figure 36), the name shape can be dragged from the box and dropped into the world. If the box is not minimised as in Figure 33 any of the shapes can be dragged from the shape list or from the name shape box and dropped into the world.

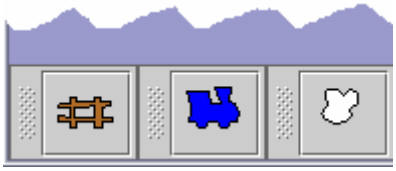


Figure 36 – Minimised performer boxes.

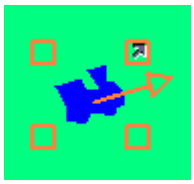


Figure 37 – A selected performer in the world.

The shape given to the performer after it is dropped into the world is the shape that was dragged from the performer box.

When a performer in the world is selected by clicking on it, it displays some orbiting controls that can be used to modify the size and orientation of the performer as in Figure 37. There are four squares in the corners surrounding the performer. Dragging one of these squares either stretches or shrinks the performer centred on its current location. There is also an arrow pointing in the direction the performer is heading. The heading is the direction the performer will move if it is given a move instruction; it is usually the same as the direction the performer is facing. By dragging the heading direction arrow, the performer can be turned to head in any direction. If the user holds the shift key down while dragging the heading arrow, the performer's heading is changed but the direction it is facing is left unchanged. This is useful when demonstrating rules for performers that move around the world but are always supposed to stay vertical for example.

Performers can be placed anywhere in the world, heading in and facing any direction and with a variety of sizes. Icicle currently has a display size limit for each performer of ten times the original size in order to stop any one performer covering the whole world. A world covered by one performer looks indistinguishable from an empty world of the same colour. This restriction is useful for beginners but there should be a way of removing it for experts; sometimes it is useful to have performers grow to any size.

Selected performers can be moved by dragging, even while a program is running. This can be used in games such as Pong, where a bat performer can be moved around by the user to hit a ball performer. When a selected performer is dragged the resize and turn controls disappear to give a cleaner appearance to the program. Usually the user does not want to see these when playing a game. Performers can also be resized and rotated while a program is running, adding to the liveliness of the environment.

DEFINING RULES IN ICICLE

Producing an Icicle program is done by defining rules for Icicle performers. We have seen how performer templates are given shapes and that performers can be dragged into the world window from the performer boxes, now we need to give the performers some behaviours.

With *interaction computing*, rules can only be produced when an interaction occurs, even though they can be edited at any time. This means three things:

- You can only make rules for a performer that is currently in the world.
- The world must be going (an interaction is not detected if the world is not running).
- The interaction (the condition for the rule) must happen in order for Icicle to construct the rule.

As an example, imagine a user wants to put a space-ship performer in the world and have it move in a straight line across the world. First, the user must make a performer. This is done by creating a new performer box, then selecting and modifying a shape to represent the spaceship. Then a space-ship performer needs to be dragged from the box and dropped into the world. If the world is not running the world must be started by either clicking on the control to make the world go (Figure 38), or by clicking on the forwards button that makes the world go through a single tick of the world clock.

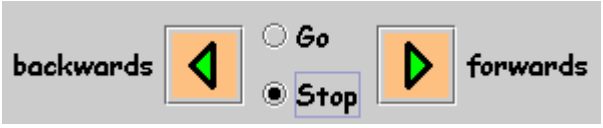


Figure 38 – The world action controls.

If the world is already running when the performer is dropped into it, or the go or forwards controls are activated, the world stops because there is no default rule for the spaceship performer and a rule editor appears asking the user to make a rule for the spaceship (Figure 39). The spaceship is also automatically selected, showing the grow and rotate controls around the shape.

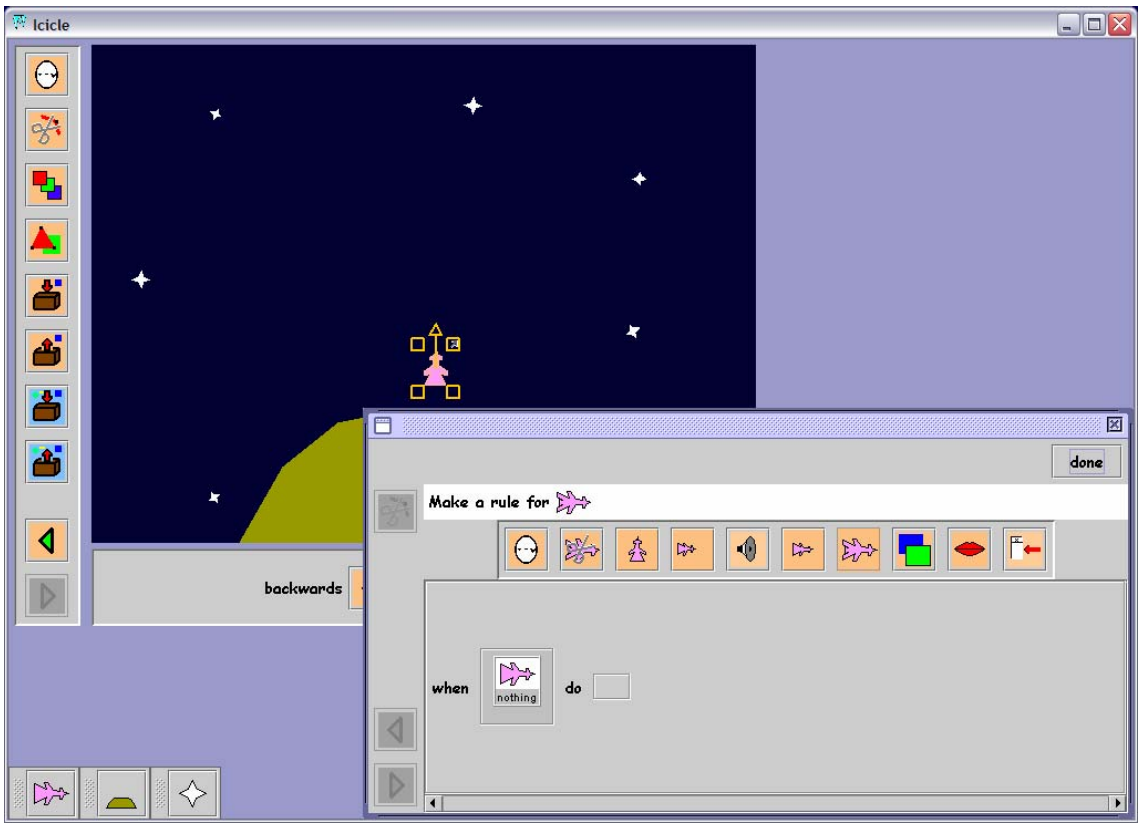


Figure 39 – The rule editor and the selected performer.

The user then demonstrates or selects the behaviour he or she wants the spaceship to have when it is not handling any interactions. If the user wants the spaceship to remain stationary and not do anything, he or she clicks the "done" button in the rule editor. The "done" button is also pressed when the instructions for the rule are complete. This closes the editor. If the world was going, rather then single-stepping, it immediately resumes without further action from the user and the performer executes its newly defined instructions, if any.

Every performer type has a default behaviour. This is known as the "when nothing is happening to me" rule, which is abbreviated to the "nothing" rule. Originally, the default rule was called the "go" rule because the default rule is what a performer usually does when it goes. Calling it the "nothing" rule gets across a bit more information about the role of the rule. It was believed that calling it the "default" rule was inappropriate for the younger age range of Icicle users.

The "nothing" rule is the rule a performer executes if and only if there are no other rules currently executing for the performer. This gives the "nothing" rule special characteristics.

- The first, is that the rule automatically repeats if no other rule is executing for the performer. As long as there are no interactions that apply to the performer, the rule repeats.
- The second, is that as soon as an interaction occurs for the performer the default rule is halted, regardless of where it is up to in its list of instructions, and the rule corresponding to the interaction is performed instead.

These characteristics seem to match the most needed behaviours and the expectations of non-programmers. Non-programmers assume that if an object is moving it will keep moving without

further intervention (Pane et al., 2001). The "nothing" rule commonly has a performer either really doing nothing or repeatedly moving around the world. As soon as something happens to the performer a new rule is executed and the "nothing" rule is stopped. Normally the "nothing" rule will be started up again when the new rule finishes.

Stopping the world for a rule

Originally, the same process of asking for a rule was repeated when any interaction occurred that did not already have a rule defined for it. However, user testing (see Chapter 7) found that stopping the world for all of these interactions and asking for instructions was disruptive, even if the program could continue with a single click on the "done" button. The annoyance experienced by users fits in nicely with the predictions made by the Attention Investment model (Blackwell, 2002; Blackwell & Burnett, 2002) with regards to cost and pay-off. If users do not want to create rules in these situations, there is no pay-off in forcing their attention to be removed from what they are currently doing.

Unfortunately, Icicle does not include a user intention prediction model and cannot tell beforehand with 100% accuracy, which rules users will want to add instructions to, and which rules they will want to leave empty. Instead, some reasonable decisions can be made a priori about the interactions that require instructions, e.g., when a key is pressed by the user it is reasonable to assume the user intends a performer to react to it, rather than to do nothing. On the other hand, it was observed that some of the interactions were hardly ever given instructions by children using Icicle; these no longer cause the world to stop and ask for instructions. These interactions are still added to the list of rules when they occur, but by default, they have no instructions to perform. If users want to provide instructions for one of these conditions, they can do that by opening the corresponding performer box and directly editing the rule.

Table 3 shows the interactions handled by performers, describes when the interactions occur and whether they cause the system to ask the user for a decision if no existing rule matches the situation. It also includes notes as to how a particular interaction can be used and special information about accessing data in other variables. This will be explained later in the chapter.

Key press interactions

Key press interactions are different from other interactions in that when they first occur no performer is associated with them. They are generated when the user presses a key. Any time the world is running and the user presses a key, the system checks to see if any performers have a rule that responds to that key. If no performers have such a rule, the world stops and the user is asked to choose a type of performer to receive the key press. Figure 40 illustrates that query when the user has typed a "w" and then clicked on a human shaped performer in the world.

Interaction	Occurrence and notes	Asks user immediately
created	when a performer is dropped into the world by the user or when performers are loaded from a saved world *This can be used to set up properties before the performer does anything else.*	no
created by	when a performer is created by another performer *Instructions in these rules have read access to the variables of the creator performer.*	no
nothing	when there are no other ordinary performer rules that are currently active *This is the* usual *repeat forever interaction.*	yes
key pressed	when the particular key is pressed by a user	yes
collides with	when a performer collides with another performer *Instructions in these rules have read access to the variables of the other performer in the collision.*	yes
no longer touches	when a performer was overlapping another performer but no longer does so	no
told (star message) by	when a performer receives a message *Instructions in these rules have read access to the variables of the performer that sent the message.*	yes
told (star message) by me	when a performer receives a message from itself *This is used to start or continue a thread e.g., to loop continuously.*	yes

Table 3 - Performer interactions.

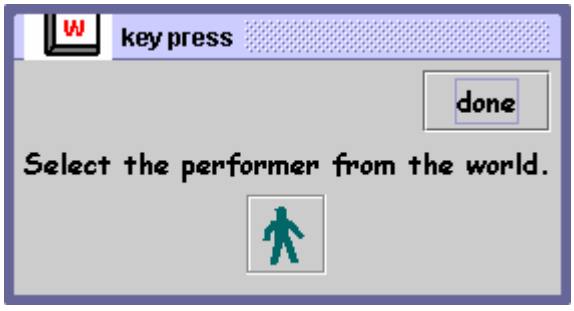


Figure 40 – Choosing the key press destination.

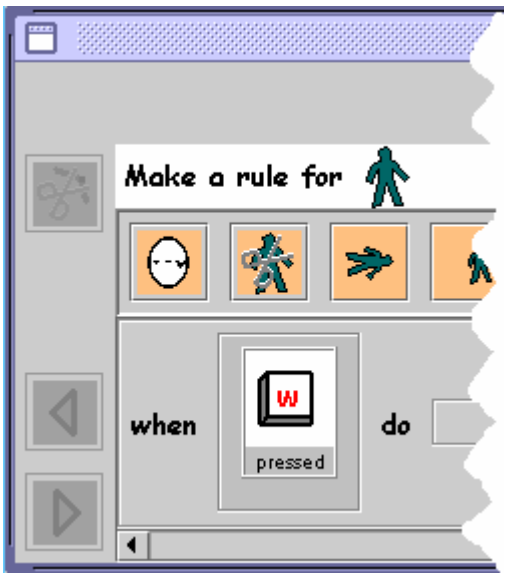


Figure 41 – Making a key press rule.

When a performer has been selected, the usual request is made to define the rule corresponding to the interaction as in Figure 41. The animation for the interaction shows the key moving in and out to signify being pressed.

After the rule has been defined, whenever a "w" is pressed all performers of the selected type execute the corresponding instructions. If the user wants one particular performer to react, he or she makes a special performer type and ensures that only one performer of that type is in the world at a time. This is not the problem it may appear. If the user is controlling one performer with key presses, that performer is special and there is usually only one performer of that type in the world anyway. If the user is making a two-player game with two performers that are both supposed to react to key presses, then the two performers will be of a different type as they will have to respond to different key presses anyway.

It is also often the case, that the user will want several performers of the same type to respond to the same key press – in particular when deleting all performers of a type or when resetting all performers before restarting a game.

Message interactions are exactly the same as key press interactions in the sense that they broadcast to all performers of the same type in the world. This is nice from the point of view of consistency and since most users use the key press interactions before needing message interactions it helps them understand how message passing works.

Defining collision rules

Collisions are special interactions because it is possible that two different types of performer are involved in the one interaction. This means that two different rules need to be defined. In this case Icicle pops up a two-rule rule editor as in Figure 42. Each of the rules is defined separately. If a performer collides with another performer of its own type, only one rule is defined. More sophisticated techniques are required if the user wants both performers of the same type to react differently.

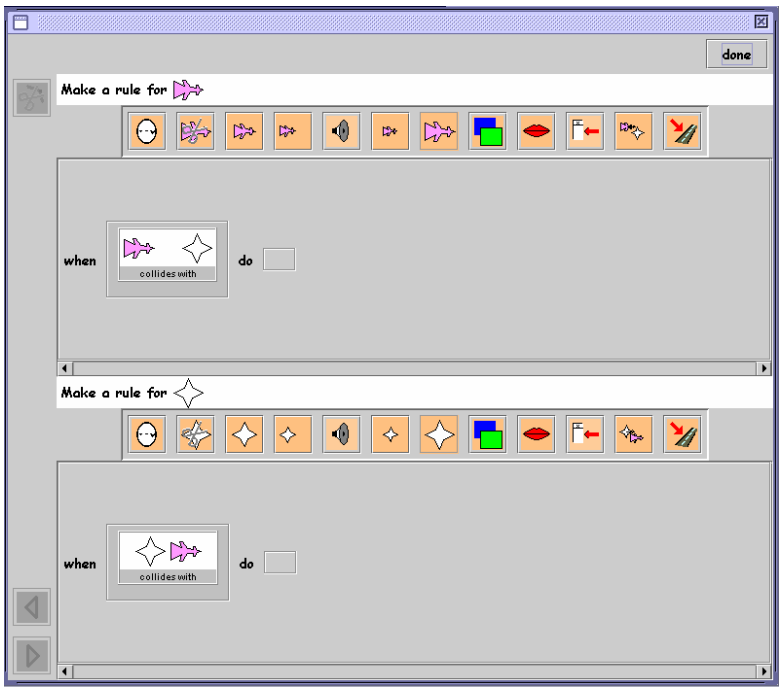


Figure 42 – Defining collision rules, one for each involved performer.

Programming by demonstration in Icicle

When a rule is being defined, instructions can be conveyed to Icicle in two ways: demonstration by direct manipulation and by clicking on command buttons. Using the earlier example of the spaceship, we will consider how the two techniques can be used to make the spaceship move. Remember that when the editor box requesting the rule appears on the screen (as in Figure 39) the Icicle world stops running and the performer associated with the interaction is selected in the world. No other performers currently in the world can be selected while the rule is being defined.

The user wants to make the space-ship move up the screen. To achieve this by direct manipulation the user drags the spaceship up the screen. As with the turtle in ICE, while the performer is being dragged Icicle does several things.

- The pointer changes to a copy of the performer being dragged.
- The selected performer is left in its original position and a dashed line extends from this performer to the current position of the dragged pointer (Figure 43).
- If the performer is dragged at a different angle from the direction it is facing the pointer rotates to show the performer facing in the direction it is being moved (Figure 44).

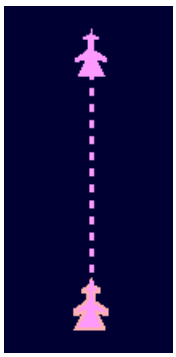


Figure 43 – Dragging to demonstrate a move.

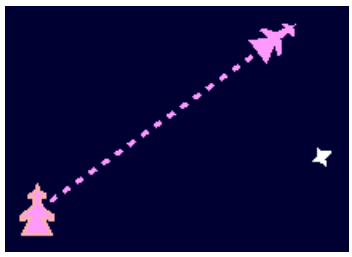


Figure 44 – Dragging to demonstrate a turn and a move.

When the performer has been moved as the user wants, the mouse button is released and the drag is terminated. This causes an instruction corresponding to this move to be inserted in the result area of the rule editor. In the case of the move shown in Figure 43 the rule editor now looks like Figure 45. As discussed in Chapter 4, the instruction is represented as an animation with a title in English – "move 157", for move forward by 157 pixels, in this case.

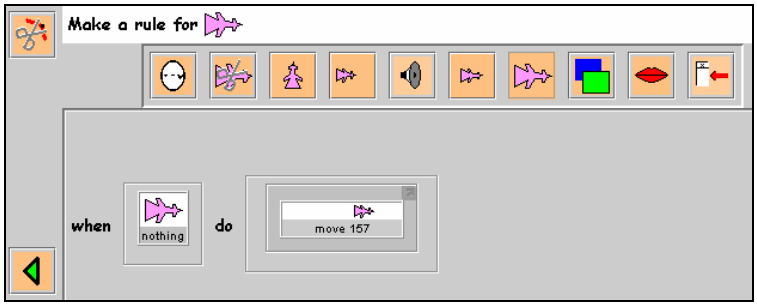


Figure 45 – The rule editor with the move instruction from Figure 43.

The performer in the world is then moved to its new position so that additional instructions can be given to it when the drag is completed. This way a series of moves can be included in the instruction by repeatedly dragging the performer as in the time-lapse diagram in Figure 46. This sequence of drags is turned into seven instructions, consisting of moves and turns, shown in Figure 47. A drag of a performer normally turns into two instructions, a turn followed by a move, as in ICE.

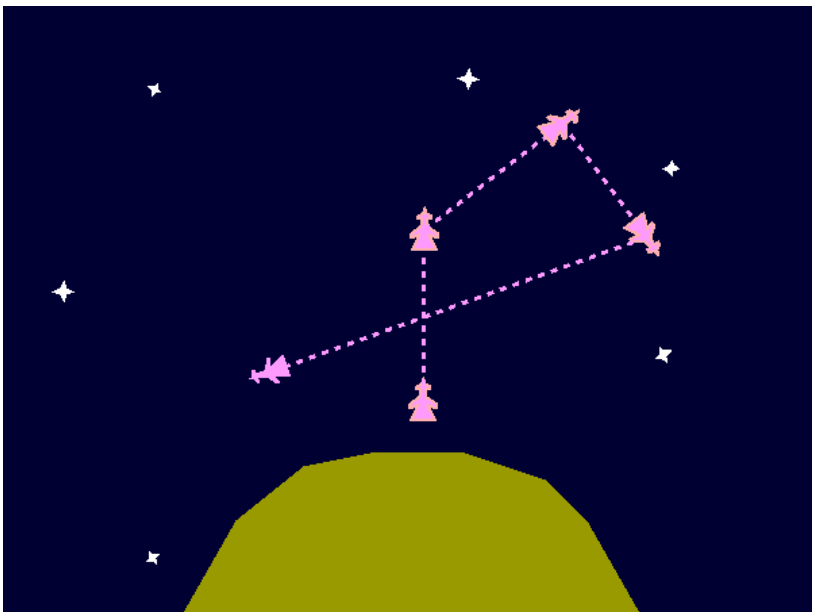


Figure 46 – A sequence of drags by the user.

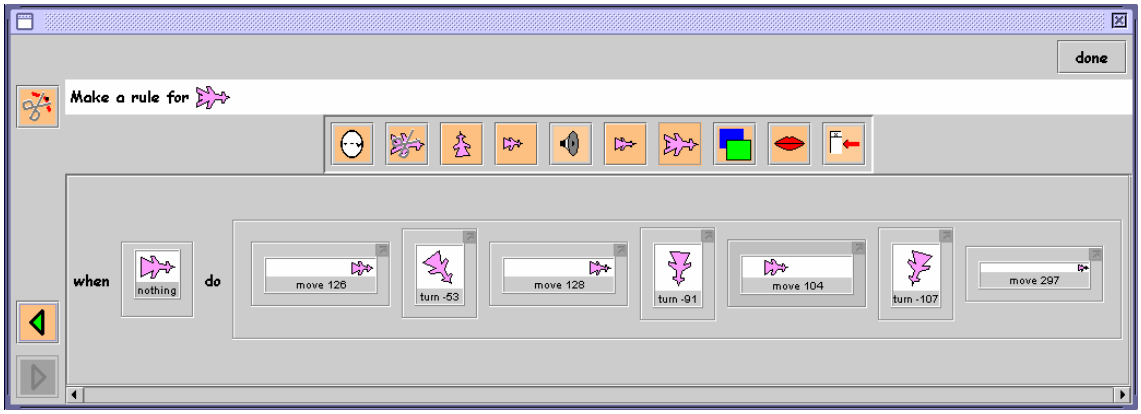


Figure 47 – The instructions corresponding to the drags of Figure 46.

If the user wants a turn without a move, this can be demonstrated by dragging the arrow control representing the direction the performer is facing (Figure 37). Similarly, a zoom instruction, to grow or shrink the performer is demonstrated by dragging one of the four squares surrounding the selected performer. Most of the other instructions can be demonstrated using direct manipulation on performers in the world as well. Table 4 shows all instructions that can be demonstrated this way and describes what the user has to do to demonstrate them. The two instructions that require shift-drags are only needed occasionally, e.g., they are useful to keep performers that represent people standing upright as they move around the world.

Instruction	Method to convey by demonstration	Notes
turn & move	drag the performer	Normally a drag causes two instructions: a turn and a move.
turn heading only & move	shift-drag the performer	The facing direction stays the same.
turn	drag the heading arrow of the performer	Turns both the heading and the facing directions of the performer.
turn heading only	shift-drag the heading arrow of the performer	The facing direction stays the same.
zoom	drag one of the size boxes	Current limit of 10x normal size. Drags larger than this stop at the limit.
morph	drop the shape to morph to on top of the performer	Care must be taken to drop it on the performer otherwise this is interpreted as a create instruction.
delete	select the world's scissors tool and click on the performer	Exactly the same as deleting a performer when not demonstrating a rule.
create another performer	drop another performer anywhere on the world	The heading and facing directions of the new performer are initially set to the facing direction of the creator performer.

Table 4 – Instructions that can be demonstrated.

Direct command selection

Just as in ICE, Icicle provides an alternate method to teach instructions when creating or editing a rule. The user can directly select commands by pressing buttons along the top of the rule editor window (Figure 45). Apart from buttons that correspond to the instructions in Table 4 there are some buttons that allow the execution of commands that cannot be demonstrated by direct manipulation of the performers. Some of the buttons cause an instruction to be inserted immediately in the result area of the rule editor: the "delete", "turn", "move", "zoom", "bring to top" and "turn to this performer" instructions. The last one only appears when the rule is the result of a collision or the reception of a message. Other buttons cause a pop-up list to appear: the "create a new performer", "make a sound", "morph this performer", "tell a performer", and "set a variable" instruction buttons. When the user has selected an option from the pop-up list, the corresponding instruction appears in the rule editor.

Several of the instructions take a parameter. The "turn", "move" and "zoom" instructions require two parameters, an angle, distance or percentage parameter and a speed parameter. The instructions that do not produce pop-up lists have default parameters so that the users are not forced to enter parameters, e.g., the "turn" instruction has a default value of turning 30° anti-clockwise with a speed of 5. Speeds for instructions range from 1 (the slowest) to 10 (the fastest). This information is summarised in Table 5.

Instruction	Start icon	Icon animation (on pointer over)	Pop-up list / default parameter
create a new performer		The egg opens to show a chick.	A list of all performers appears.
delete this performer		The scissors close and the performer fades.	
turn this performer		The performer rotates anti-clockwise.	A default turn of 30° anti-clockwise with a speed of 5.
move this performer		The performer moves repeatedly backwards and	A default move of 50 pixels with a speed of 5.
make a sound		The speaker has lines representing sound coming	A list of all sounds appears. These are represented by descriptive titles, such as "fret noise" or
zoom this performer		The performer repeatedly grows and shrinks.	A default zoom of 150% with a speed of 5.
morph this performer		The performer repeatedly morphs through all of its	A list of all shapes appears.
bring to top		The performer appears on top of the squares.	
tell a performer		The mouth opens.	A list of all star messages this performer can send appears.
set a variable		The arrow moves to the left.	A list of all this performer's variables appears.
turn to this performer		The upper-left performer turns to face the other performer.	

Table 5 - Icicle instructions.

The buttons representing instructions have the icons shown in Table 5. In order to improve the closeness of mapping between the buttons and their actions, the icons are generated as the rule editor opens, from the performer that the rule is being created for – in Table 5 that is the spaceship performer. These icons animate to show the effects of the instructions when the pointer is moved over the buttons as in Baecker et al. (Baecker, Small, & Mander, 1991). The buttons also produce text tooltips when the pointer hovers over the button.

When a rule is produced, any instructions either demonstrated or selected directly have an immediate effect on the performer in the world associated with the rule, as well as appearing simultaneously in the rule editor. Demonstrated instructions can be interleaved with directly selected commands, e.g., the user might drag the performer across the screen to generate a turn and a move instruction and then click on the morph button and select a shape to morph to. The result on the performer in the world is that it has moved and had its shape changed and the rule editor shows animations of the three instructions.

In the same way, if the user undoes an instruction in the rule editor, not only is the instruction animation removed from the rule editor, but also the performer in the world has the effect of that

instruction undone. In the example above if the user pressed the undo button after selecting the morph instruction the performer in the world would still be in its new position but it would return to the shape it had before the morph instruction was carried out.

THE ICICLE INSTRUCTIONS AND EDITORS

Many instructions demonstrated by direct manipulation need their parameters refined in order to do exactly what a child requires. Similarly, instructions given by pressing on command buttons have default parameters that need modifying. To accommodate these changes most of the Icicle instructions have editors associated with them. The instruction editors allow the precise setting of the parameters for the instructions. The instruction editors appear when children click on instructions in the rule editor with the secondary mouse button (usually a right-click). This section describes each of the Icicle instructions along with their associated editors.

Move, turn, zoom and play a sound

The *move*, *turn*, *zoom* and *play a sound* instructions are similar and have both a change parameter and a speed parameter.

- *move*: The performer moves through a number of relative pixels, either forwards (positive) or backwards (negative).
- *turn*: The performer turns through a number of degrees, either anti-clockwise (positive) or clockwise (negative).
- *zoom*: The performer grows or shrinks. The change parameter is a percentage of the performer's current size, so a value greater than 100 enlarges the performer and a value less than 100 causes the performer to shrink.
- *play a sound*: The performer plays a sound, from the standard list of Java MIDI samples. The change parameter is the pitch.

Move instructions are relative to the size of the performer. A larger performer moves further and faster when it executes the same move instruction than a smaller one does. This is useful when simulating three-dimensional movement (see Chapter 8) as larger versions of performers look and act as though they are closer than smaller performers of the same type.

These instructions share an editor, shown in Figure 48. This allows the user to type an exact change value and to vary the speed the instruction operates at by dragging the slider. One is the slowest speed and ten is the fastest. The only difference between the *move* editor and the *turn*, *zoom* and *play a sound* editors is the description before the number entry box, "move" in the case of Figure 48. For the *play a sound* instruction, the speed parameter is the volume control.

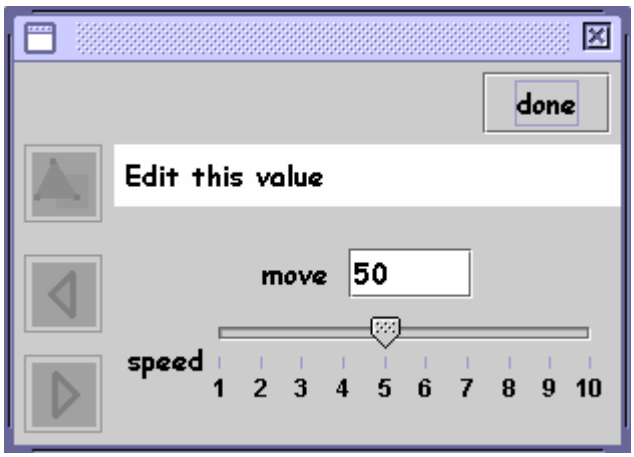


Figure 48 – The editor for a move instruction.

The *morph* and *delete* instructions should also have a speed editor, but this has not yet been implemented. The *bring to top* and *turn towards* instructions do not have editors.

Creation

The performer creation instruction is very powerful. When a performer is created, it can be positioned relative to the creator performer with a relative orientation and size.

If a *create performer* instruction is demonstrated (see Table 4) the new performer is dropped where the user wants it to be, relative to the performer executing the instruction. The benefits of relative positioning were learnt from ICE. Usually, performers are created quite close to the creator performers and in a particular spatial relationship with them. In some ways, this captures the strength of the grid system in Creator, where a character is created relative to a particular layout of characters, but without its limitations.

In Figure 49 the performer doing the creating (the square) is selected, the performer being created (the thin rectangle) has been dropped above and to the right of the square. By default, any performers created by dropping are created at the same relative size and heading as the creator performer. This means that if the creator performer is large the created performer is correspondingly large. Because the precise placement of created performers can be crucial to a program, the relative position, size and heading of created performers can be altered with the creation editor. This will be described shortly.

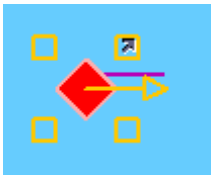


Figure 49 – Creation by dropping.

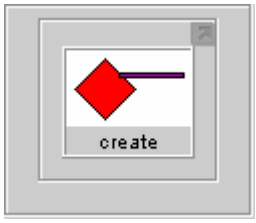


Figure 50 – A create instruction.

Figure 50 shows the instruction in the rule listing that was produced by the drop illustrated in Figure 49. The animation of the instruction shows the created performer, the thin rectangle in this case, gradually appearing, even though in the running program the creation is instantaneous.

Icicle's technique of providing parallelism within instructions will be described later in this chapter. This technique can be used to create several performers simultaneously. Figure 51 illustrates the situation in the world when four performers have been dropped around a creator performer. This gives rise to four instructions as in Figure 52, note how the create instructions maintain a closeness of mapping to their actions by showing the created performers relative to the creator. By dragging three of these instructions and dropping them on the fourth, the sequential instructions become parallel. This means that all four performers will be created at the same time. The instruction in the rule listing becomes like that shown in Figure 53.

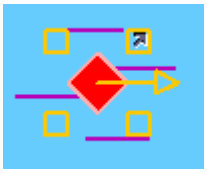


Figure 51 – Multiple creation by dropping.

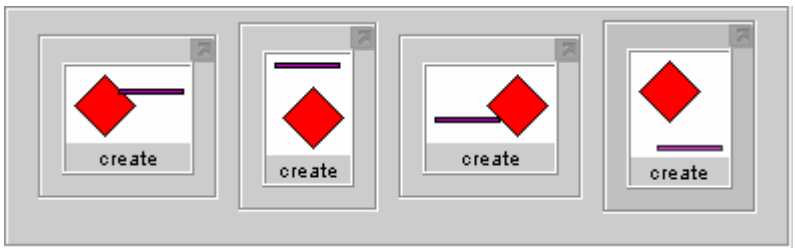


Figure 52 – Multiple sequential create instructions.

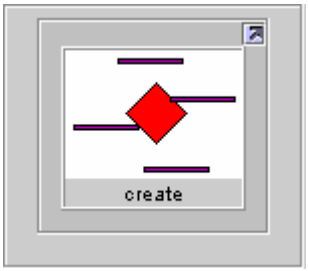


Figure 53 – Multiple parallel create instructions.

The creation editor

The creation editor is the most sophisticated instruction editor. This allows the relative placement, size and heading of a performer to be adjusted using the same dragging actions and controls that are used when placing performers in the world. This editor was inspired by the difficulty, mentioned in Chapter 2, that children had in positioning and orienting drawings produced by procedures in ICE.

If the instruction represented in Figure 50 is right-clicked on it opens the creation editor window as in Figure 54. In this editor, the performer created by the instruction is selected and can be resized,

moved and rotated. Figure 55 shows one possible configuration after editing and what the corresponding instruction looks like.

Observe that in Figure 55, not only has the created performer, the thin rectangle, been enlarged relative to the creator performer, the diamond, but the creator performer looks as if it has been enlarged as well. This is because the editor window automatically scales its contents in order to provide the best possible complete view of all the performers it contains. In this way, the created performer can be dragged a long way away from the creator performer and still be carefully positioned relative to it.

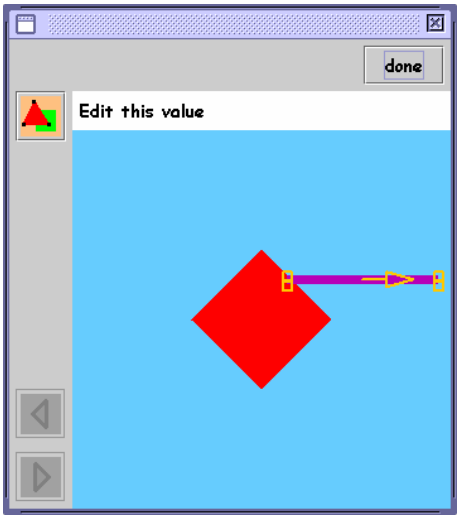


Figure 54 – The creation editor.

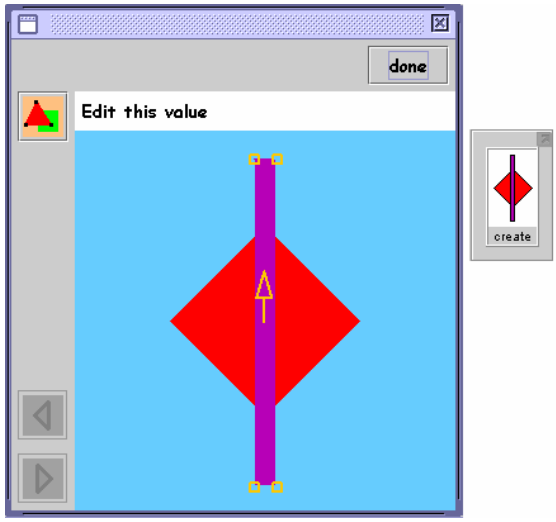


Figure 55 – After moving and resizing and the corresponding instruction.

If the created performer is dragged a long distance from the creator, the editor provides a rectangle indicating the size of the Icicle world to give the user some perspective on how the instruction will appear in the world as in Figure 56. If the creation instruction was initially demonstrated by dropping the created performer in the world, it could have been placed anywhere relative to the creator.

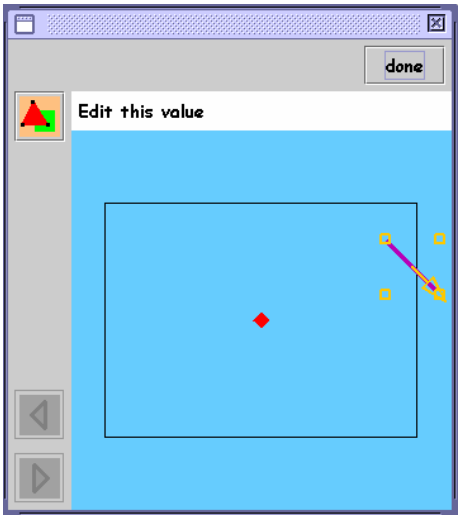


Figure 56 – Creation at a distance.

Figure 57 – Parallel creation editing and the corresponding instruction.

When several performers are created in parallel, the combined creation instructions can be edited in the same editor. This is very useful as the created performers can be positioned relative to each other as well as to the creator performer. The layering of the created performers can also be changed in the editor. Using this technique very complicated instructions can be produced. Figure 57 shows a teddy bear construction instruction, this consists of 14 performers created in parallel and positioned by the creation editor. The execution of this instruction creates a teddy bear in the world. By positioning, resizing and turning the creator performer, the teddy bear can be positioned anywhere and be of any size and face any direction when it is created. This creation of complicated objects is one of the powerful Icicle idioms and demonstrates the usefulness of positioning objects relative to each other.

Deletion

The instruction to delete a performer adds some visual interest by fading the performer out as it leaves the world, over a number of clock ticks. The animation that represents this in the rule listing shows the performer gradually disappearing. As soon as a performer begins executing a delete instruction, the performer is effectively no longer in the Icicle world, even though its fading appearance may still be there for a few ticks of the clock. In particular, any new interactions that would have affected the performer are ignored.

Messages

The messages in Icicle are represented by pictures and are called star messages. Each message type is represented by a picture of a star with a certain number of points and a certain colour. Figure 58 shows the drop down menu a user gets after clicking on the "tell a performer" button. Each performer type in the world is represented in the drop down menu; including a special type, "Me". In this example, there are three types of performers in the world: octagons, triangles and squares. The user selects the type of performer the message is to be sent to, and then a number of possible messages appear on the right-hand side. In the case of the octagon, the only message that can be sent is a five-pointed red star. In Icicle, this is referred to as a "red 5" message. As well as presenting all existing messages for a particular performer type to the user, Icicle always generates one new message name. This means that in the example in Figure 58 there are currently no existing messages to the octagon, the "red 5" is the new message type.

When the user selects a message type the instruction is represented by a *tell performer* instruction (Figure 59). The instruction shows the type of performer the message will be sent to, in this case an octagon, and the type of message being sent, a "red 5". Like all Icicle instructions, it animates when appropriate; the mouth opens and the box around the star is highlighted.

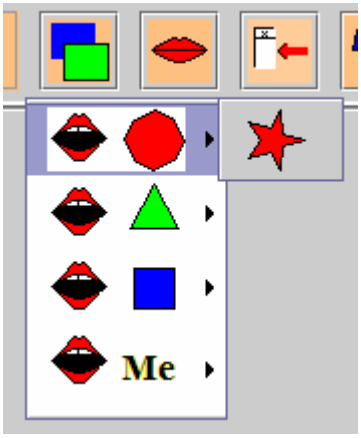


Figure 58 – Drop down message selection.

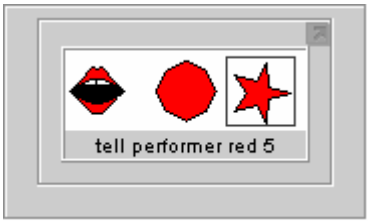


Figure 59 – A tell performer instruction.

The next time the user wants a performer to send a message to an octagon the selection of messages will include the "red 5" message plus a new message type as in Figure 60. Each newly generated message is allocated a colour by stepping sequentially through seven colours; when all colours of five-pointed stars have been used, the number of points on the stars increases by one and the colours are reused. Even though this technique becomes unworkable after too many different message types – it is difficult to tell the difference between a ten-pointed star and an eleven-pointed star, in reality it has never been a problem. Most Icicle programs employ only a small number of message types.

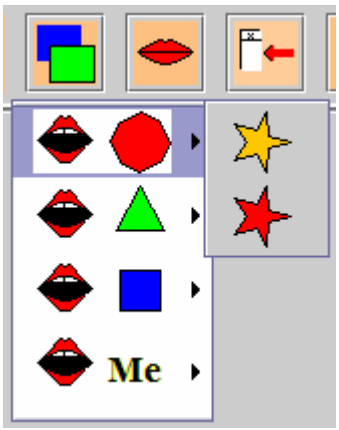


Figure 60 – Message selection with two choices.

Receiving a message is an interaction. This is represented in the rule condition as a "told message by" animation (Figure 61). The message is particular to the type of performer that sent it. In the case

of Figure 61 the "red 5" message was sent by a square performer. Along with collision and creation interactions, message reception interactions start rules that have read access to the variables in the other performer involved in the interaction – in the case of message reception, to the variables of the sender performer. This is how extra information rather than merely notification of an event is conveyed by the message.

Figure 61 – Message reception interaction.

Telling oneself

In the drop down list of performers that appears when the tell instruction button is pressed (see Figure 58) there is a special item called "Me". This enables a performer to send a message to itself. Sending a message to itself is one way for a performer to start a new thread of instructions. An infinite loop is produced if a "tell me" instruction is the last instruction of the rule that responds to that same message. Figure 62 shows two such rules. The top one makes a square performer repeatedly move in a square path with sides of 200 relative pixels. The bottom one repeatedly sends a message to another performer; how this can be used, we will look at next. Both of these loops could be started when the performer is created by sending the original messages from a "created" or "created by" interaction.

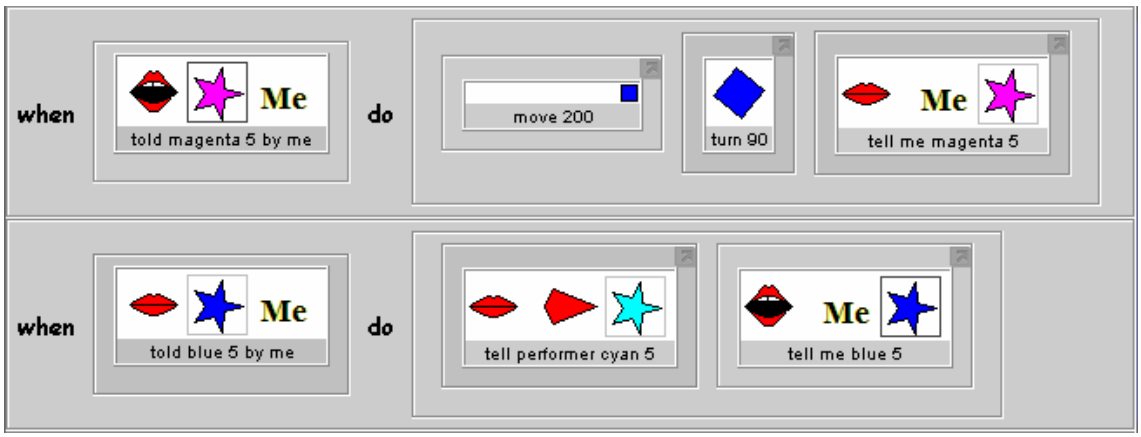


Figure 62 – Infinite loops using messages.

Turning towards another performer

The *turn towards* instruction is different from the other instructions in that it refers to two performers – the performer the instruction is for and the performer that it should turn to face (see Table 5).

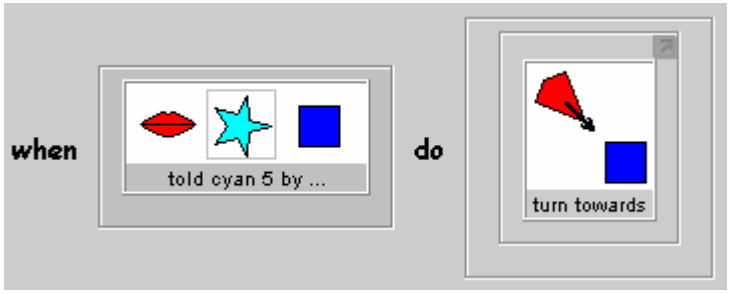


Figure 63 – Turning to face the sender of the message.

In order to identify the performer to turn towards there has to be some reference to that performer in the rule condition. The only interactions that refer to other performers are those associated with collisions (and their opposite, *no longer touching*), the interactions that deal with handling messages sent by other performers, and *created by* interactions when one performer has been created by another. Obviously if the rule interaction is a collision, there is knowledge of the performer this one has collided with, so the *turn towards* instruction can be used in a collision rule. Similarly, and more importantly, any message receiving rule has knowledge of the message sender, and so a performer receiving a message can turn towards the performer that sent the message. In Figure 63 the red arrowhead shaped performer receives a "cyan 5" message sent by the square performer in Figure 62 and turns to face the square. Since messages are broadcast to all performers of the same type these rules can be used to make several arrowhead performers continually turn to face the square as it moves (Figure 64).

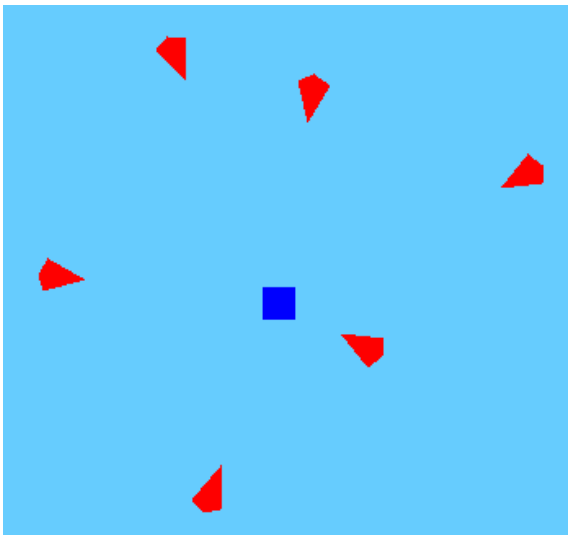


Figure 64 – Multiple performers facing the message sender.

Parallel instructions[1]

Every performer in the Icicle world operates independently of all other performers until some interaction occurs between them. Each performer moves around the screen according to its own stream of instructions, in parallel with all other performers.

Icicle also allows more than one rule to be executed by each performer at the same time. In addition, when demonstrating or editing the instructions of a rule, Icicle allows the user to combine individual instructions so that the performer can carry out more than one instruction, simultaneously, within the one rule.

For example, in the case of the dog running across the screen in Chapter 4 (Figure 29) we really want the dog to change shape as it moves. In Icicle, this is as simple as dragging instructions on top of each other.

The child starts by demonstrating the behaviour sequentially. It does not matter whether the morph is demonstrated first or whether the movement across the screen is demonstrated first.

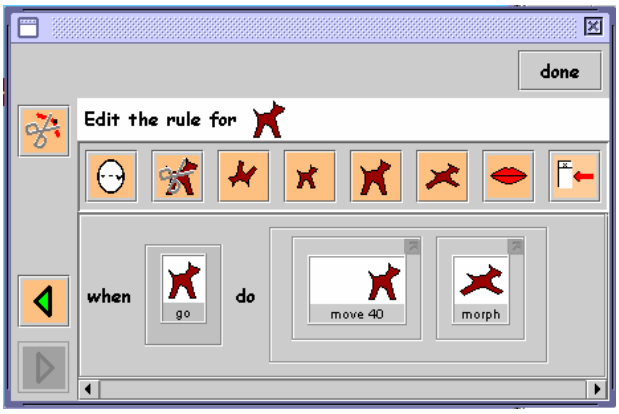


Figure 65 – The rule editor showing a move and a morph instruction.

These instructions then appear as two animations in the rule editor (Figure 65). The user then drags one of the instructions and drops it on top of the other. In Figure 66 the morph instruction is being dropped on top of the move instruction.

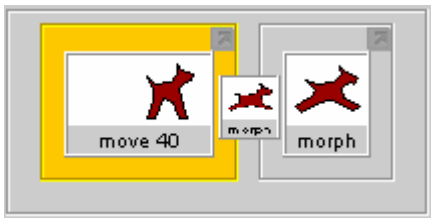


Figure 66 – One instruction being dropped on another.

This immediately turns the instructions into one parallel instruction and the animation shows the combined instruction, the dog changes shape as it moves (Figure 67). The combined box is called a parallel box.

[1] Portions of this section appeared in (Sheehan, 2003b)

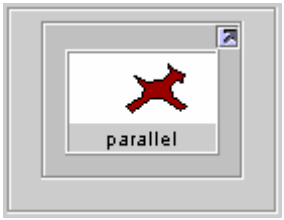


Figure 67 – A parallel instruction of a move and a morph.

This is not the only way the parallel instruction can be represented. By clicking on the arrow in the top-right hand corner of the animation the parallel box expands to reveal all of the instructions it contains (Figure 68).

There are no limits (except taste) on the number of instructions inside a parallel instruction. In some situations, such as initializing many variables, all can be done at the same time in one parallel instruction.

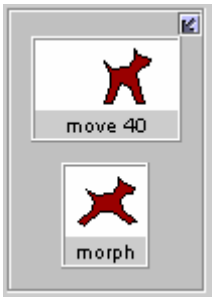


Figure 68 – An expanded parallel box.

This simple technique adds tremendously to the types of action that can be represented in an Icicle program.

Working with variables

Variables are accessible from the variables tab of the performer box. Figure 69 illustrates this. The values shown in the variable boxes are those of the selected performer. Extra variables can be created for each type of performer by pressing the "create new variable" button on this tab. Variables can be deleted using the scissors tool. It is impossible to delete the four standard variables: across, down, heading and zoom. Currently only numeric variables are supported by Icicle. They are floating point values rounded in the display to two decimal places. This was a compromise. Originally, the system made them appear like integers and only integers could be entered into them, but this made some programs more difficult to produce. Two decimal places were chosen rather than showing very long strings of digits after the decimal place.

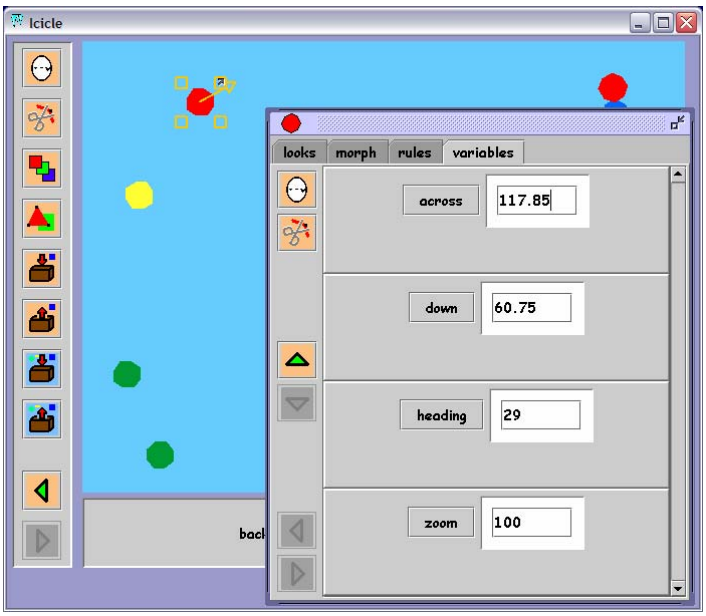


Figure 69 – The standard variables.

The variable assignment instruction is selected by clicking on the "set a variable" button (see Table 5). This produces a pull-down list of all variables this performer holds. Selecting one of these adds a set variable instruction to the rule list, as in Figure 70. In this example, the set instruction will increment the "score" variable by one. Any reference to a variable in the expression reveals the current value of that variable when the pointer moves over the reference; compare Figure 70 with Figure 71. The influence of Boxer (DiSessa & Abelson, 1986) is apparent in this and in the nesting of boxes used to represent the instruction.

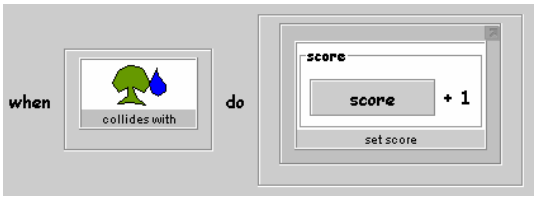


Figure 70 – Incrementing a variable.

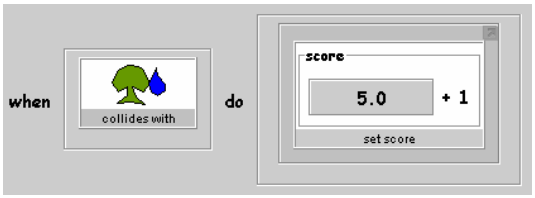


Figure 71 – The variable value displayed.

In the simplest of cases, a variable may simply be used to contain a value and not produce any instructions, e.g., in the example just described, the program adds one to a "score" variable when a raindrop collides with a tree. The first time the collision occurs and the variable assignment is carried out, the user is prompted with a request to define an action for the "score" variable (Figure 72). In this case, the user does not want any action caused by the assignment statement and so, he or she leaves the rule with no instructions.

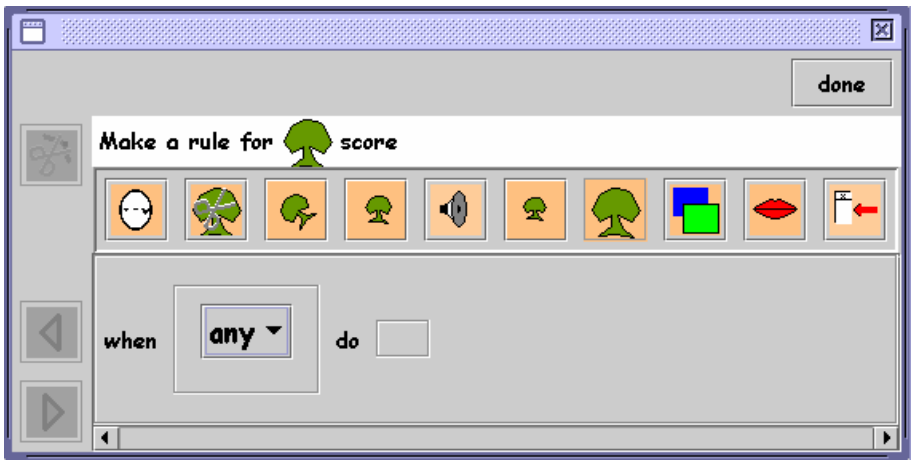


Figure 72 – Default variable assignment condition.

The variable assignment rules are different from ordinary performer rules because the condition, the bit between the "when" and the "do", can be modified. The default condition is to accept any value. So the rule in Figure 72 should be read as "when *score* is assigned any value do nothing". Like all rule editing windows this one shows the performer the rule is associated with in the instruction buttons and in the prompt line – "Make a rule for". In the case of variable rules, the actual variable name is included in the prompt line as well. We shall see how variable rules interact with ordinary performer rules in Chapter 6.

Interaction	**Occurrence**	**Asks user immediately**
< number or variable	when the value assigned is less than the number or variable	n/a
<= number or variable	when the value assigned is less than or equal to the number or variable	n/a
= number or variable	when the value assigned is equal to the number or variable	n/a
> number or variable	when the value assigned is greater than the number or variable	n/a
>= number or variable	when the value assigned is greater than or equal to the number or variable	n/a
!= number or variable	when the value assigned is not equal to the number or variable	n/a
any	when the value assigned is anything	yes *This is the default condition when a value is assigned to the variable the first time.*

Table 6 - Variable interactions

The complete range of variable interaction types is shown in Table 6. The "any" condition is the default and can be changed by the user pressing on the "any" pull-down list. Any of the comparisons can then be selected, illustrated in Figure 73. The rule might be true for a limited range of values, e.g., if the score is less than 10 do something, but use another rule when the score is greater than or equal

to 10. The user can both select a comparison and enter a value, or select a variable with which to compare the assignment value.

Since the completed rule must correspond to the condition that caused the request for the rule to appear in the first place, the rule editor employs some intelligence to ensure that the condition, after editing, matches the value that was being assigned to the variable. For example, if the variable was being assigned the value 6 and the user changed the condition to be "<", the value displayed would change to 7. The condition would now be "when *score* is assigned a value < 7". This still matches the original condition; being assigned 6 matches a rule that checks for a value less than 7. Similar automatic changes occur for all possible comparisons. If the user changes the value, 7, in this example, the comparison in/equality may be automatically altered. In this case if the user types a 4, the comparison changes to ">", since 6 is greater than 4. The rule editor only allows valid numbers to be typed into the comparison field. These transformations of the interaction are to ensure that the user cannot make a mistake. The rule condition will remain true for the actual value being assigned to the variable.

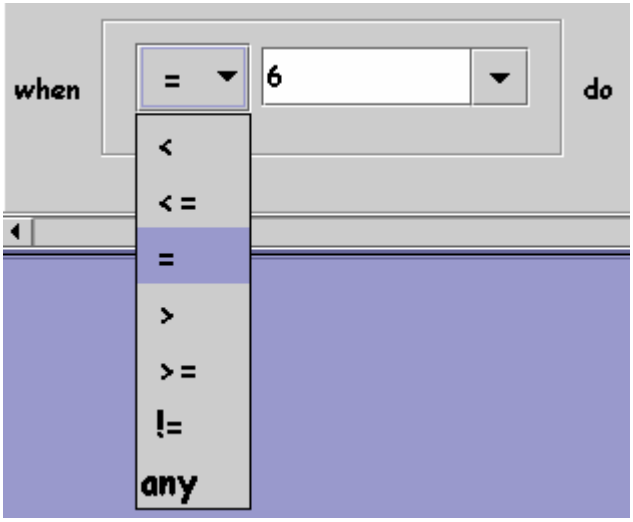


Figure 73 – Assignment conditions.

Average example

An example to compute the average of a set of numbers entered by the user should demonstrate how variable interactions work to perform simple calculations.

First, since all computations are carried out by performers in Icicle, a performer has to be created. The performer is dropped in the world in order to run and the user gives the performer a new variable called "next number". With the world running the user types 10 into the "next number" variable and presses the enter key. This causes a rule request window similar to that in Figure 72 to appear.

Because Icicle is lively and interactive, new variables can be created at any time, even when a rule is being defined. The user creates two more variables "total" and "count". By default, all new variables have a value of zero. The user makes two assignment rules, one to increment the "count" variable, and one to add the new value of "next number" to the "total" variable. After adding an extra variable

for the average, the rule is completed with an instruction to divide the "total" value by the "count" value and store that in the "average" variable (Figure 74).

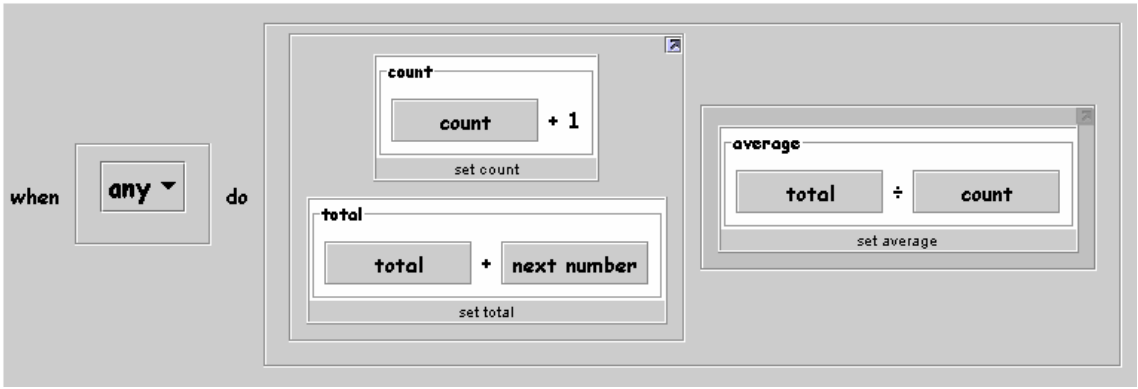


Figure 74 – Calculating an average.

Compare this calculation of the average with the Logo version in Table 7. The major reason the Icicle version is simpler is that input and output do not have to be handled explicitly by the program. Icicle, like Boxer, allows values to be entered directly into boxes representing variables. Similarly, the values in the boxes are automatically updated when the variables change.

```
to average
    calculate_average 0 0 next_number
end

to calculate_average :total :count :number
    if :number < 0 [print :total / :count stop]
    calculate_average :total + :number :count + 1 next_number
end

to next_number
    type [Enter the next number (-ve to finish):]
    make "number readword
    print :number
    output :number
end
```

Table 7 – Calculating an average in Logo.

To calculate the average of the numbers 10, 6, 19, 22, 11, 4, 12 and 8, the user types the remaining numbers in turn in the "next number" variable and presses the enter key. The "count", "total" and "average" variables show the running totals and results as each value is entered. Figure 75 shows the final values.

Rules for the standard variables

Instead of asking novice users of Icicle to define rules for the four standard variables, "across", "down", "heading" and "zoom", as soon as they have values assigned to them, Icicle provides default rules. The rules for "heading" and "zoom" are trivial, nothing happens. Of course, they can be edited by the user to do something interesting when they get new values. For example, the user can keep a

performer heading in a certain direction by resetting its "heading" variable to the original value whenever a different value is assigned to it.

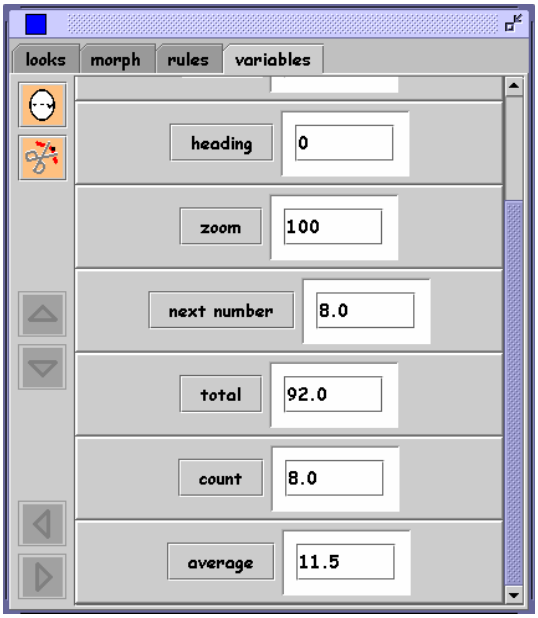


Figure 75 – The results of the average calculation.

Most versions of Logo have two modes for turtles moving off the screen, either the turtle really does move off the screen and cannot be seen until it moves back or the turtle wraps around – if it moves off one edge of the screen, it immediately reappears at the opposite edge. Icicle has no need for a low-level, built-in, wrap-around mode for this situation. This is handled by the variable interaction mechanism and the default rules for the "across" and "down" variables. Figure 76 shows the rules for "across". If the value assigned to "across" is less than 0, it is replaced with 599; the world is 600 pixels wide. Remember that these values are live. As soon as the "across" value is changed the performer jumps to that new position. Similarly if the value is greater than or equal to 600 it is replaced with 0. Users can modify or delete these rules to constrain the performers to smaller sections of the screen or to remove wrap-around all together.

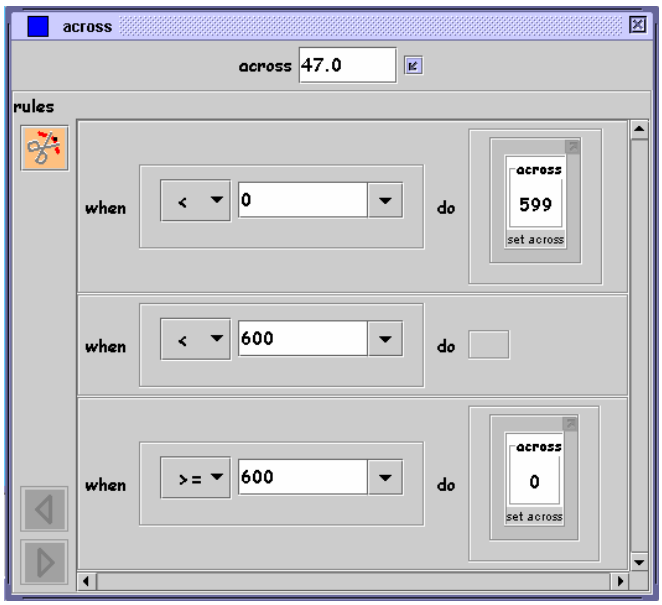


Figure 76 – The default rules for "across".

The rule conditions for variables are checked in the order they appear in the variable window. The "value < 600" is only checked when "value < 0" is not true. Only one rule is active for each variable at a time. This is different from normal non-variable rules as will be discussed in detail in Chapter 6.

Referring to variables from another performer

As has been stated several times, an Icicle performer cannot directly modify the state of other performers. Only the rules of the owning performer can be used to assign values to that performer's variables. Any variable rules are treated as rules of the performer with which the variable is associated. You can see this in Figure 72 where a rule is being created for the "score" variable of a tree performer. The instruction buttons refer to the tree performer, so the variable rules are a special subset of the rules associated with a performer.

The send message instructions can be used to tell another performer, or a group of performers, to modify their own state. Sometimes in this situation, it is necessary for the receiving performers to get information about the state of the sender. We have already seen this with the *turn towards* instruction; the receiving performer must have a reference to the position of the sending performer in order to turn towards it.

Any receiving performer has read access to the contents of any of the sending performer's variables. This way messages can be used to pass information between performers – the sender puts the information into a local variable, it sends the message, and the receiver accesses (but cannot alter) the data in the sender.

The same applies for collision type interactions. There are two of these the "collides with" and the "no longer touches" interactions (see Table 3). In both situations, the performer responding to the interaction can access information about the other performer involved in the interaction. The same also applies to the *created by* interactions. The created performers can access information in the performer that created them.

As an example, consider a situation where a triangle performer is sometimes chasing a PacMan performer and sometimes not, depending on the state of a "chase" variable in the PacMan performer. Figure 77 shows one way this can be done. The PacMan performer sends a message to the triangle. On reception of the message, the triangle turns towards the PacMan and checks to see if it should chase the performer by testing the value of the PacMan's "chase" variable. It does the testing by assigning the value to one of its own variables, in this case called "test". The "chase" variable of the PacMan is represented by a box with the name shape of the performer and the variable's textual name.

The triangle performer then either does nothing, if the value of the "chase" variable was one (as it is already facing the PacMan), or it turns to face the opposite direction if the value was zero. These rules for the "test" variable are shown in Figure 78.

This ability to test the values of variables in other performers provides the basis for more complicated programs. In Chapter 8 this is used to sort performers according to values in their variables and to simulate Turing machines.

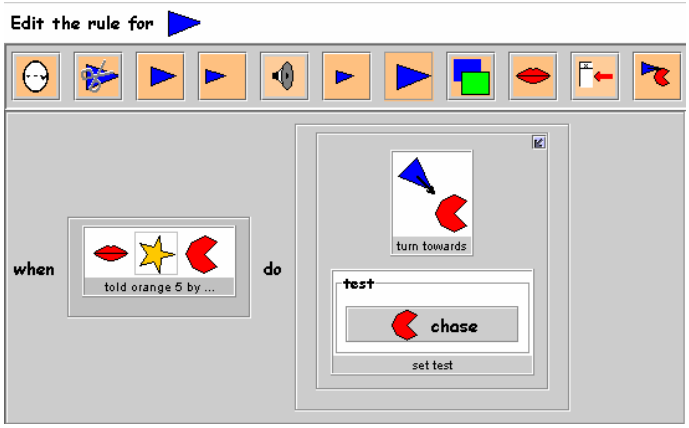


Figure 77 – Accessing a value from the sending performer.

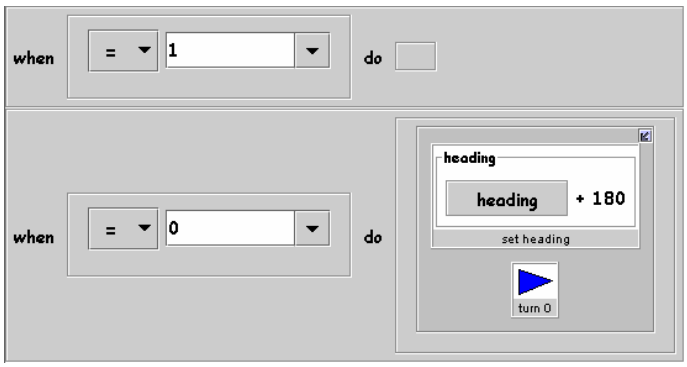


Figure 78 – The rules for the "test" variable.

The variable assignment editor

No mention has been made of how the expressions are assigned to variables as in Figure 74. When a *set a variable* instruction is entered in the instruction list, it has a default value of zero. To modify this, the user right-clicks on the instruction and an expression editor appears.

The assignment expression editor (Figure 79) allows simple arithmetic expressions to be created using the four basic operators and references to constants, variables of the executing performer, random numbers, and variables of the sending performer. The editor is designed to make it impossible to enter a syntactically incorrect expression. The operators and value panels alternate in being active. Therefore, expressions must be of the form:

value operator value operator …

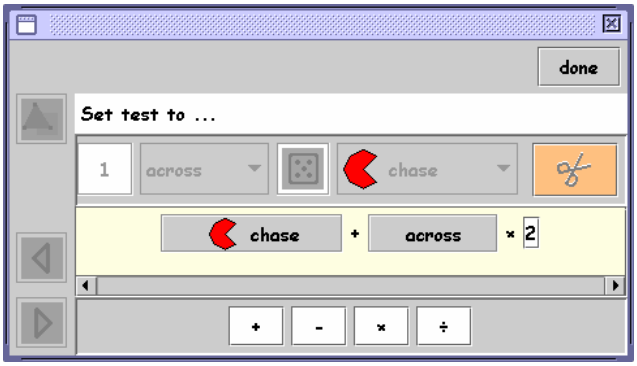


Figure 79 – The assignment expression editor.

Any dangling operators are ignored when the "done" button is pressed. The editor does not currently provide parentheses but expressions do use the normal algebraic order of evaluation.

REDUCING THE NUMBER OF RULES

In Chapter 4 it was mentioned that graphical-rewrite systems like Stagecast Creator sometimes require a very large number of rules to capture behaviour. The transition to rules based on interactions with continuous motion and position in Icicle, has the added benefit of reducing the number of rules required to convey program information. For example in a train simulation only two rules are necessary to make a train move along the tracks (Figure 80): one rule to move forward and one rule to tell the train to face the centre of the piece of track it has just collided with. These simple rules are adequate for moving over reasonably complicated track layouts as in Figure 81. The equivalent program in Creator (Figure 82) would require many more rules because Creator requires specific placements and orientations of characters, as well as specific appearances for them, in order for the before part of a rule to match. In AgentSheets the number of rules is reduced by adding semantics to the rewrite rules (Repenning, 1995) e.g., trains can be constrained to flowing along tracks.

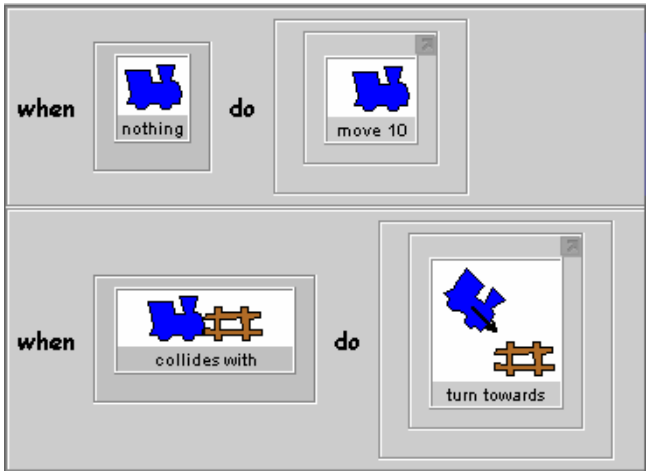


Figure 80 – The two rules required to move the train on the tracks

The train program distributed with Creator actually uses the names of characters to cut down on the number of rules used. For example, a piece of track that switches from going across the screen to travelling either up or down is called "switch up down left" and the movement rules for the train check whether the characters beneath the train and adjacent to it contain the words "up" or "down". These are no longer graphical rewrite rules.

Because of Icicle's ability to position performers in any orientation, the Icicle version has only one shape for the train, compared to the four appearances in the Creator version, and there is only one performer class with one shape for all pieces of track, compared to eight different characters for track in the Creator version.

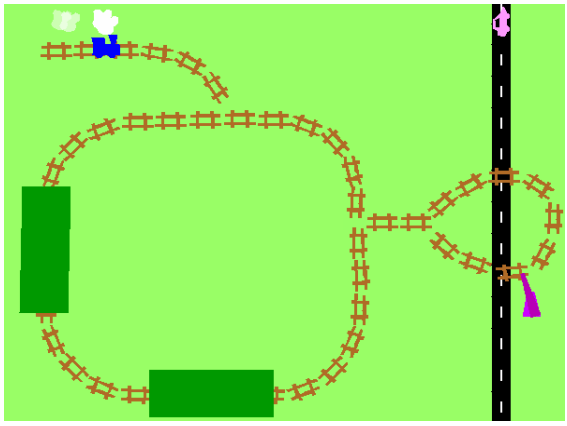


Figure 81 – An Icicle train world.

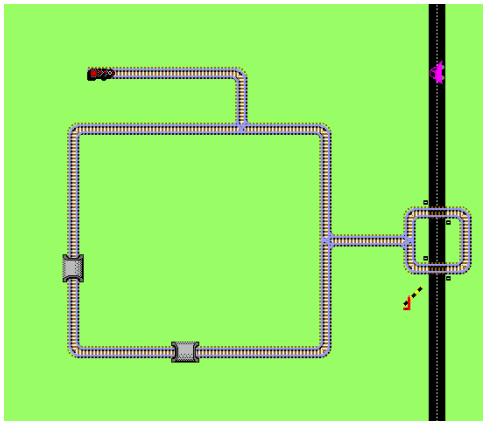


Figure 82 – A Creator train stage.

To be fair, there is a difficulty with the Icicle version of the train set. The tracks have to be placed carefully to ensure that the train collides with them. If the train does not hit the next section of track, it will go off the rails. This becomes tricky in situations such as switches. In this case the tracks must be positioned in such a way that the train will collide with the section of track you want the train to follow. The train set in Figure 81 sends the train off to the loop on the right-hand side only when travelling clockwise. If the train is going around anticlockwise, it does not collide with the piece of track leading to the loop.

THE RULES PANEL

The rules for a performer type are stored together under the rules panel of the performer box (Figure 83). Moving the pointer over a rule in this panel causes the rule interaction and instructions to animate in sequence. Clicking the mouse button when over a rule opens the normal rule editor, as in Figure 39, and allows the direct selection of instructions, as well as the editing of instructions. Instructions can be moved around in the editor by dragging and dropping them in the required position. Instructions can be deleted by selecting the scissor tool and clicking on an instruction. As in other situations in Icicle, these changes can be undone and redone in case of mistakes.

Unlike the rules for variables, the order of rules in the performer rules panel is irrelevant to the selection of which rule to make active. As we shall see in Chapter 6, all current interactions cause the corresponding rules to activate. They can all work in parallel. This is different from Creator and AgentSheets where the first matching rules are normally chosen. This makes the conceptual model of rule selection simpler in Icicle, at least for the common cases (i.e., normal rather than variable rules).

Originally, Icicle allowed rules to be dragged around in the rules panel, to be ordered as the user desired. In practice, this was never done, except by me, and so the rules were presented in the order they were defined. Unfortunately, this led to a jumble of rules, as they were defined in the order they occurred. In addition, many of the rules did not do anything; they had no instructions. This happened, either when the user pressed "done" when asked for the rule, or the interactions were ones that the user was not asked to describe a rule for.

The current solution is to apply heuristics to the rules so that they appear in a more useful order. The first heuristic is that empty rules, ones without any instructions, appear at the bottom of the list, except for empty "key pressed" or "told by" rules. The reason for the exceptions was that rules of these types only arise because of decisions by the user; the interactions do not occur unless the user wants them to. The second heuristic is, of the rules with instructions, the "created" and "nothing" rules appear at the top because they happen first or regularly. The user provoked rules, "key pressed" and "told by", come before collision rules. Also if both the "collides with" and its corresponding "no longer touches" rules have instructions they are grouped together. In this case, the rules are likely to be linked, e.g., the "collides with" rule might add one to a variable and its matching "no longer touches" rule might subtract one from it. Table 8 summarises the rule order according to interaction and whether the rule is with or without instructions.

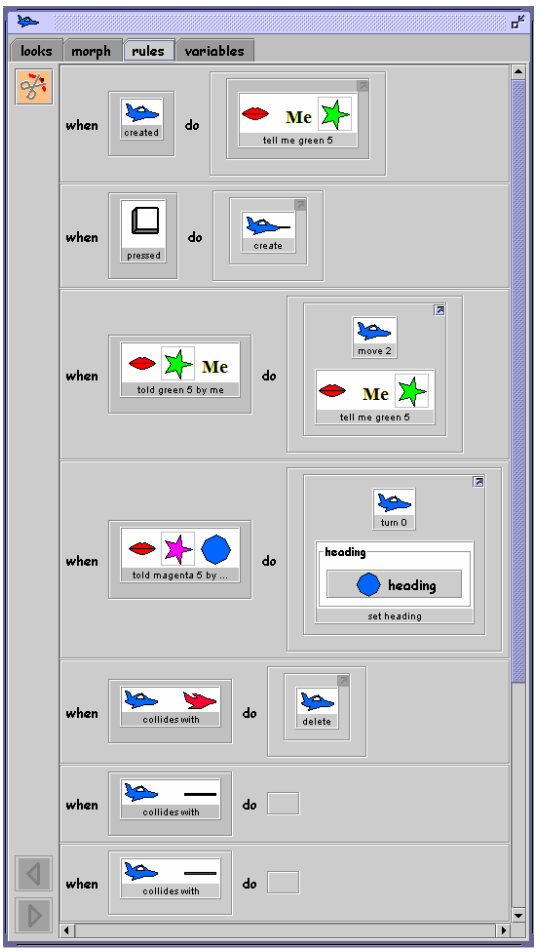


Figure 83 – The rules panel of a performer box.

The rules for variables are accessible from the variables panel (Figure 75). Clicking on a variable name opens a variable window showing the current value of the variable (this field can be altered by the user) and the variable's rules, as in Figure 76.

With instructions
created
created by
nothing
With or without instructions
key pressed
told (star message) by
With instructions
collides with
no longer touches (grouped with matching collision)
Without instructions
collides with
no longer touches
created
created by
nothing

Table 8 – Display order in rules tab.

Chapter 6

IMPLEMENTATIONS

In this chapter, some of the implementation details of Icicle are revealed. In doing this, I examine some operating system concepts, such as scheduling performer threads and dealing with parallelism, and explain how Icicle deals with potentially huge numbers of interactions.

The last part of the chapter covers another sort of implementation – the implementation, within Icicle, of common programming patterns, including loops and conditionals.

JAVA

Icicle is implemented in Java. There were two reasons. The cross-platform nature of Java meant that Icicle could run on almost any modern machine: Apple Macs[1], Microsoft Windows machines and Linux boxes. Most schools use either Macs or Windows machines. The other reason, was that Java was the programming language, including accompanying libraries, that I knew best. Icicle was to include a large amount of user-interface code (more than half of the source code of the current version) and the Java user-interface classes were the only ones with which I had experience.

Later in this chapter, the speed of Icicle is discussed. Part of the speed problem could be attributed to Java but overall the current Java virtual machines are perfectly adequate, as can be seen by the excellent implementations of NetLogo and Creator in Java.

SCHEDULING

Icicle must schedule many rules simultaneously. Each performer can have several rules running at any moment. In Chapter 4, parallelism in four other environments (ToonTalk, Creator, AgentSheets and NetLogo) was described. How do these environments implement parallelism and how does Icicle implement it?

Selecting instruction streams

Table 2, on page 75, includes the description of instruction streams in the five environments. All of the environments have to select instruction streams for scheduling. This entails two parts, analogous to medium-term and short-term scheduling in operating systems.

The first part is concerned with collecting the instruction streams that are available for execution. In ToonTalk, a robot is available for scheduling when it is presented with a box containing data that matches its parameters. A Creator or AgentSheets rule is ready for scheduling when the state of the program matches the conditions of the rule. NetLogo is more traditional; a block of code is made available for scheduling by requesting that it runs, e.g., by *asking* all turtles to do something. Icicle rules are selected when an interaction occurs.

[1] Apparently, Apple Computer has officially changed the name of the computer line from Macintosh to Mac.

The second part deals with choosing which instruction stream, of all those available, should be running now. All of the environments maintain a list of instruction streams that is processed in order. Creator and AgentSheets execute their instruction streams in the order they were detected as active, and run them to completion before executing the next stream. The other three environments are more complicated and allow parts of each active instruction stream to be interleaved with parts from other streams. The amount of processing before switching varies.

The unit of scheduling

On most home and school computers we still have single processors, so all talk of parallelism is really at the level of breaking streams of instructions into small pieces and jumping quickly between the instructions from different streams in such a way that the user perceives the result as many different streams running simultaneously. In operating systems, these are called "time-slices" or "quanta". Because they are not always related to time in the environments discussed here, I will refer to these as "chunks". In these environments, things can be quite different from switching between processes (or even threads) at the operating system level. A chunk can be a logical unit of code rather than a varying number of instructions allocated according to a time-slice. The reason for this is that the scheduler that allocates the chunk size can be designed for the particular class of instruction streams in the environment. In most operating systems, the scheduler has no information about the meaning or relevance of the instructions it is currently selecting from each process or thread. Whereas, in this type of environment, the scheduler has access to information as to which streams are associated with which objects and, more importantly, the instructions themselves are stored as higher-level constructs than machine instructions. For example, the instruction presented to the scheduling system may be "forward 50", which actually equates to hundreds (if not thousands) of machine-level instructions. This means the instructions can be broken up into sensible chunks by the scheduling system, to maximise parallelism without compromising efficiency.

In this case how big should the chunks be? The smaller the chunk, the smoother the parallelism will be. The usual consequence of a very small chunk size is that a lot of time is spent switching between the instruction streams. The consequence of a very large chunk size is that the instruction streams do not really appear to be running in parallel.

The ToonTalk scheduling mechanism is multilevel. Any robots operating in the foreground (visible on the display) execute to completion (unless they block waiting for a result). In this case, a chunk size is the length of the code associated with one robot, or method. Robots executing in the background use time-slices and pre-emptive scheduling (Kahn, 2004). In that case, the chunk size depends on the length of the time-slice.

Creator deals with chunks the size of a rule body. When a rule is executed, all instructions in the rule complete before anything else happens on the Creator stage. In one Creator time period there may be several rules associated with one character that will be executed, but they will be executed sequentially rather than in parallel. By turning on the global attribute, "moving at the same time", no intermediate moves of the characters are displayed. The only visible state change is the final state of all

characters at the end of the time period. This has the effect of hiding the fact that the rules executed sequentially and speeds the world up. When executing normally, without the "moving at the same time" attribute, Creator displays all changes to characters. For example, in one rule, a character may move to several different squares and change its appearance several times; each of these changes is normally visible.

AgentSheets also schedules complete rules at a time. One rule is executed to completion before another rule executes even if they are both scheduled. The worksheet (the equivalent of an Icicle world) does not always update after every rule; the window updater appears to be running in a thread separate from the thread executing agent rules, and there is no fine control over this behaviour.

Netlogo switches between instruction streams when the stream makes a "change to the world", such as moving a turtle or setting an agent's variable. Setting local variables has no effect on the scheduler. This facilitates calculations; a turtle can do a lot of internal work, which is carried out very quickly, and only loses its use of the processor when it does something visible.

Because the chunk lengths in these schemes are usually very small to a human the illusion of parallelism is strong, but the number of machine instructions per chunk is very large and hence efficiency is not a problem.

In Icicle, all instructions are broken up into microinstructions that complete in one clock tick. A single instruction may be broken up into many microinstructions, e.g., the instruction to "move 100" may be broken up into 20 microinstructions of "move 5". The size of the microinstruction parameter depends on the speed of the move instruction, the greater the speed, the larger the move. Every active rule has an instruction stream of microinstructions associated with it. One microinstruction from each active instruction stream is executed every clock tick (see Figure 84). In this way, multiple rules behave as if they are really executing in parallel.

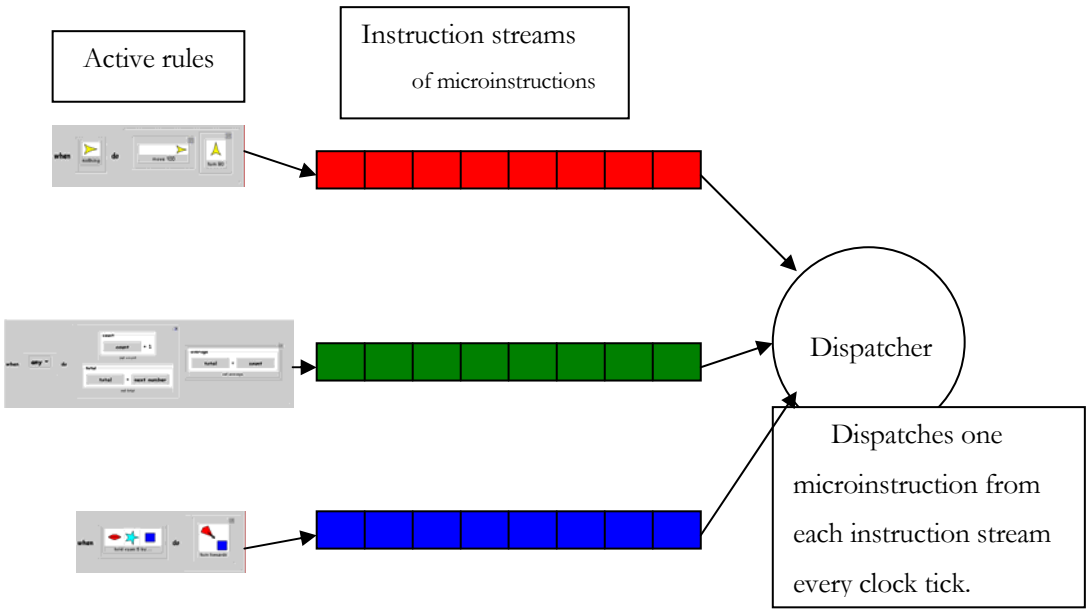


Figure 84 – The Icicle scheduler.

This form of scheduling has very little overhead. When a rule is activated, a stream of microinstructions is added to the list used by the dispatcher. If the stream is empty, because it has not run before, the associated rule instructions are converted into a stream of microinstructions. After this, the dispatcher selects one microinstruction from each instruction stream in turn and invokes it. Some of the microinstructions, e.g., sending a message, cause interactions to occur. These are collected throughout the clock tick. At the end of the clock tick, other interactions are checked for, such as collisions. If any interactions do not have rules associated with them, either the user is prompted for the rule instructions, or an empty rule is created for the interaction. The instruction streams produced by the new interactions are added to the list feeding into the dispatcher.

The Icicle scheduler can easily be converted to run on a multiprocessor. Each instruction stream can be allocated to its own processor. There would be some extra overhead in checking for collisions as this would be have to done after every move, turn, zoom or morph microinstruction on each processor, and there would have to be global synchronization to stop all instruction streams when a rule needed to be defined by the user, but the basic scheduling scheme fits the requirements of scheduling on a multiprocessor.

Parallel instructions

In Chapter 5, we saw how instructions, in the same rule, can be combined to run in parallel by dropping one instruction on top of another to create a parallel box. None of the other environments provides a way to do this.

Commonly, the instructions in a parallel box need a different number of world clock ticks to execute. In the example with the dog, the move instruction might take eight clock ticks to complete whereas the morph instruction might take five clock ticks to finish. When the instructions are combined, the faster instruction is split and stretched out evenly over the time needed by the slower instruction. This guarantees that the changes will progress together and finish at the same time.

Sound instructions work nicely in parallel instructions. The sound continues over the period of time the other instructions require to complete. The sound begins at the first tick of the parallel instruction and finishes at the last tick.

Not all instruction types can be split this way. The ones that can are those that take more than one clock tick to execute: moves, turns, morphs, zooms, making sound and deletions. The performer creation, message sending and variable assignment instructions take only one clock tick to complete. In these cases, the single clock tick instructions are guaranteed to run in the first tick allocated to the combined parallel instruction along with the first ticks of all of the other instructions in the same parallel box.

One of the innovative features of the Icicle scheduler is that a parallel box instruction is treated as a single instruction by the dispatcher. A parallel instruction is split into a number of microinstructions just like any other Icicle instruction. Each of these microinstructions can actually be several ordinary microinstructions, all of which are executed in the same clock tick. Such microinstructions are known as parallel microinstructions. In effect, this means that a single instruction stream can have several microinstructions scheduled each clock tick, when they are part of a parallel instruction. With the

example in Figure 68, each parallel microinstruction would include a move microinstruction and a morph microinstruction.

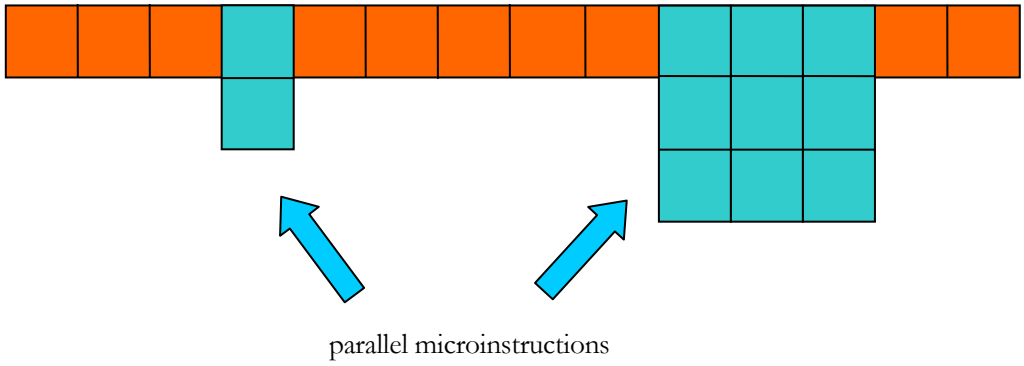


Figure 85 – An instruction stream with parallel microinstructions

Figure 85 illustrates an instruction stream as in Figure 84. It shows parallel microinstructions as columns of ordinary microinstructions. Each column of the instruction stream is taken as a single microinstruction by the scheduler and all microinstructions in the column execute in the same clock tick.

MULTIPLE INSTRUCTION STREAMS FOR INDIVIDUAL PERFORMERS

Whenever an event such as a key press or a message from another performer arrives, a decision must be made. Should the performer cease the work it is currently doing and start a new stream of instructions to deal with the new event, or should it continue with the original stream and add an extra parallel stream? Scenarios can be described to justify both approaches. If the dog, in the example from Chapter 5, hears a cat, it might stop running across the screen and bark at the cat. In this case, the first approach is applicable. If the dog starts barking after hearing the cat, and continues running, the second approach is applicable.

This second approach is the standard form of parallelism provided by Icicle. Any interaction affecting a performer, such as a collision with another performer, the reception of a message, or the user pressing a key associated with the performer immediately starts up another instruction stream in parallel with those already working for the performer. There is one exception to this, to be described shortly, the default rule of each performer.

As mentioned in Chapter 4, this appears to be a more natural model for novice users to understand than only allowing one thing to happen to a performer at a time. It also allows more accurate control for user actions, e.g., if a key press causes a performer to turn through 30°, then three quick key presses will start three versions of the rule and the result will be a combined turn of exactly 90°. If the other approach was used, where the commencement of a new instruction stream stopped existing instruction streams on the same performer, then three quick key presses could cause a turn anywhere from 30° to 90°.

This design can be employed by a user to start up as many instruction streams as required, by instructing a performer to send messages to itself. Each message causes a new instruction stream to be added. It is even possible to start multiple instruction streams in one clock tick, by sending messages in parallel with each other.

Even though there is no limit to the number of parallel instruction streams a performer can have running at any time, in practice there are usually only a small number. The list of instructions in each rule tends to be small, and this means that most instruction streams are active for a short period of time.

The default rule

The default, or "nothing", rule is the exception mentioned above. As described in Chapter 5, rather than running in parallel with any other rules the "nothing" rule can only run if the performer has no other active rules. If any interaction occurs to the performer while the "nothing" rule is running, its instruction stream terminates immediately and the instruction stream associated with the interaction begins. The "nothing" rule starts again as soon as there are no real interaction rules active for the performer.

Synchronization

The default rule provides the most common form of synchronization required in an Icicle program. A performer moves around the world under the control of its default rule until something happens to it, then the performer deals with the interaction. The default rule is stopped immediately the interaction occurs. There are other situations where synchronization between instruction streams is required.

Deletion

If a spaceship is travelling through space in a program and hits an asteroid, the spaceship performer is destroyed. When the spaceship is destroyed, with the delete instruction, all instruction streams associated with the performer are removed from the dispatcher. Similarly new interactions cannot start new instruction streams for the performer.

The instruction streams of variables

Most instruction streams do not finish until they execute their last instruction. If we want to implement the stream stopping behaviour as the result of an event, we require a stoppable type of instruction stream.

The instruction streams associated with variables can be used for this. As we saw in Chapter 4 variables are associated with performers, i.e., you cannot have a variable without it being part of a performer, but they have their own instruction streams.

When a variable has a value assigned to it, the rules of the variable are checked for a condition matching the new value and the instruction stream of the matching rule is submitted to the dispatcher. Variable rules can do anything ordinary performer rules can do. This means that a variable instruction

stream can command the performer to do something in parallel to all of the commands coming from its ordinary instruction streams.

The main difference between variable instruction streams and ordinary instruction streams is that each variable can only have one instruction stream active at a time. The advantage of this is that it is simple to terminate variable instruction streams – assign a value to the variable. This is equivalent to a software interrupt.

We will examine some examples later in this chapter, but this is an obscure way to terminate an instruction stream. A form of syntactic sugar should be added to Icicle that uses this mechanism but presents a simpler conceptual model to the users.

EXECUTING THE MICROINSTRUCTIONS

The dispatcher has the job of sending each microinstruction to be executed. Each microinstruction is represented as a Java *method* object. These objects are produced when the rule instructions are converted into microinstructions. The dispatcher calls the *invoke* method on the microinstructions in order to execute them and maintains the list of undo instructions so that the world can run backwards. Because of concern with the overhead involved in calling the *invoke* method another version of the dispatcher was implemented using a jump table for the microinstructions. As this version was no faster, the original implementation was retained.

SPEED

Even though speed was not included in the list of guidelines that Icicle was built around, the system had to work reasonably smoothly to stop children getting frustrated with it and to make performers appear lively.

Here, two areas of the design are described that are directly related to the speed of Icicle: the rule selection method and the detection of collisions between performers. The design of the rule selection method arose naturally from the description of *interaction computing*, and the most costly interaction methods, those associated with collisions, were implemented with a quick collision detection algorithm. Unfortunately, the speed of Icicle is still an issue and some compromises have been made in order to run it on slower machines.

Rule selection

As mentioned in Chapter 4, Icicle is a production system. In most production systems, the rules are explicitly checked each cycle of execution, in order to ascertain which instructions should be executed next. This is true of Creator and AgentSheets. Many such systems then have a selection phase where one of the rules that are matched is chosen to run. The simplest such scheme is choosing the first rule that matches. Then the order of scanning the rules, looking for matches, becomes very important. Creator rules can be grouped and a variety of rule selection schemes used on the group, such as, choosing randomly from amongst the rules that match.

In production systems, each rule is checked against the state of the program, or database. As the number of rules and the size of the database increases, checking the rules for a match with the current state takes an increasingly long time.

Icicle does not check rules each cycle of execution or clock tick. Every state change that occurs during the previous cycle (the interactions of Table 3, page 90, and Table 6, page 108) causes Icicle to locate a corresponding rule and prepare to execute it. The rules are stored in a hash table with interactions as the keys. Finding the instructions is very fast; no database needs to be scanned for each rule, only interactions that have occurred are searched for. If a rule does not exist in the hash table, either the system asks the user to produce the rule for execution or it generates an empty rule depending on how the interaction has been classified.

This was one of the reasons Icicle rules were called "when" rules rather than "if" rules. When an interaction occurs, it is handled. This is different from many production systems, not only in the technique for finding rules, but also, in the way one performer can be responding to many interactions at once. There is no selection from amongst active rules for a performer.

The 2-D collision algorithm

Of all the possible interactions, the two collision interactions "collides with" and "no longer touches" were the most complicated for the system to detect. Not only can the performers be any shape but they can also be any size. A Binary Space Partitioning (BSP) Tree technique (Möller & Haines, 2002) could have been used in order to get rid of the $O(n^2)$ comparisons, but as the number of performers overlapping at any one point in the world was going to be small, a sweep-and-prune interval method (J. D. Cohen, Lin, Manocha, & Ponamgi, 1995) using AABBs (Axis-Aligned Bounding Boxes) was simpler and in this case linear. If a possible collision is found by this technique, the performer shapes are then compared at the pixel level to determine if an actual collision occurred.

Each performer maintains its own AABB that is updated as the performer moves, morphs, zooms and rotates. When a performer is added to the world, its x and y bounding intervals, beginning and end points, are added to the sorted lists of all performers' intervals. Collisions are detected at this point and when intervals change their relative positions. Every clock tick the interval lists are re-sorted. Only if interval end points are not in their original positions in the list can a change in collision status, either collided or no longer overlapping, have occurred. As long as performers do not make very large movements around the world the lists are almost sorted, and so the sorting time, and hence the collision detection time, is very close to $O(n)$.

Originally, the intention was to detect where collisions would have occurred even if performers moved past each other between clock ticks. This happens when performers are moving very fast. This was not done, but will be contemplated for a future version of Icicle.

Even with large numbers of performers in the world, the collision detection algorithm does not slow things down noticeably. Another concern could have been the number of collisions for which the user must produce rules. With n performers in a world, there are potentially $O(n^2)$ collisions, but,

as the rules are for performer types rather than actual performers, the number of different types of collision is $O(p^2)$ where p is the number of performer types. Usually p is much smaller than n.

Clock ticks

Even with the fast rule activation system, the speed of Icicle is still an issue.

Icicle executes programs in discrete intervals. This allows a user to step forwards or backwards in a program, just like stepping forwards or backwards when watching a DVD. These intervals are known as clock ticks.

During a clock tick, all active threads get a little processing done. If the total amount of computation for all threads can be completed within a clock tick, the system pauses until the end of the clock tick. If the amount of computation is greater than can be executed in a clock tick, the clock tick is ignored and all threads still complete their instructions. This has the effect of maintaining a constant speed for actions such as moves and turns as long as the number of instructions for each tick can be completed within the tick. This design means that most programs run at the same speed regardless of the speed of the underlying processor.

When there are too many instructions, the world slows down; no instructions get lost. The clock tick length was originally set at $1/50^{th}$ of a second. The development machine had an AMD Athlon 1800+ (1533MHz) processor and even with 100 performers executing threads, it was possible to get reasonable performance without an obvious slow down. Performers that are not doing anything do not take any processing time. Interestingly, when running the same programs on a machine with an 1800MHz Intel Pentium IV machine, the performance was not as good. When Icicle was taken into a school for testing with children (see Chapter 7) the machine it ran on was a 1200MHz Pentium laptop. Even with a relatively small number of parallel threads, Icicle worlds started to slow down on this machine. In order to ameliorate the transition between situations where all computations completed within a clock tick and situations where they no longer completed, the clock tick length was doubled to $1/25^{th}$ of a second. This had the effect of slowing all Icicle programs down, but increased the number of threads that could run before any change in behaviour became noticeable to the users.

THE PUZZLING NATURE OF UNDO AND REDO

As we saw in Chapter 4, Icicle provides comprehensive undo facilities. Changes made to shapes, rules, and the Icicle world itself can be undone and redone if desired. The undo system even supplies the ability to stop the Icicle world and step back through its previous instructions.

Initially there were two cases if the user wanted to step forward through an Icicle program. When the instructions for the next clock tick were stored in the redo stack, the instructions were played back. When there were no further instructions in the redo stack, the subsequent steps were executed using the usual execution mechanism.

That seemed straightforward enough, but became very difficult if user interactions were being redone. If the user had moved a performer or dropped a performer in the world and had stepped back in the world to a time before the performer was moved or added, what should the behaviour be

when the user pressed "go", or stepped forwards again? One option was to mimic the moving or dropping of the performer when the corresponding step was reached. Unfortunately this had the disturbing behaviour of making performers move, or appear out of nowhere for no apparent reason. This was especially disconcerting if the user had stepped backwards a long way in the world. The default number of clock ticks available for stepping backwards corresponded to the last twenty seconds of execution time and so it was quite easy to go back a long way (up to 500 clock ticks) and then to make the world "go" forwards from this point.

The current behaviour is that stepping forwards or running the world causes it to execute from the present state of the world and does not replay user interactions. If the users want to, they can replay their own actions at the appropriate time.

Keeping the last 500 clock ticks for stepping backwards requires a very large amount of memory. Each performer in the world can be simultaneously executing an arbitrarily large number of instructions. Each instruction is associated with an instruction stream that in turn comes from a rule the performer is executing (see Figure 84).

Keeping undo information for variable assignments constitutes a large part of the memory consumption. In a single clock tick, there can be arbitrarily many assignments to each variable. In principle, each variable has to maintain an unknown number of values in order to undo 500 clock ticks. As there can easily be hundreds of variables changed in each clock tick, the potential for severe memory allocation problems is high. In practice, only the last assignment in a clock tick has an effect in the following clock tick; all other values are lost. For undoing all of the individual assignments, the value that has to be remembered is the value stored in the variable before the first assignment in a clock tick. Rather than storing all of the unnecessary values to undo each assignment instruction, the original value is stored with a count of how many times the variable is set in that clock tick. When the individual assignments are undone, they each restore the variable to the state it was in before the clock tick began.

ICICLE PROGRAMMING PATTERNS

This section is not about the implementation of Icicle but about implementing programming solutions in Icicle. Some of these are so useful that standard patterns have been developed for them. Other patterns represent traditional control structures. We will examine some common patterns to get a feel for this type of programming.

Loops

Loops can be generated in several different ways in Icicle; each of the techniques has slightly different properties. The simplest method uses the default or "nothing" rule. Because this rule runs whenever there is no other active rule for the performer, it repeats as long as no interaction occurs to the performer that corresponds to a rule with instructions. Interactions that have empty rules do not affect the running of the default rule.

The only problem with the looping properties of the default rule is that it does stop whenever another rule is activated. If this is not the desired behaviour, a more complicated technique needs to be used.

In Figure 62, we saw a method of repeatedly sending messages at the end of a rule in order to reinvoke the rule. Once the starting message is sent this loop will continue throughout the lifetime of the performer. As demonstrated in Figure 86, this can be used repeatedly to send messages to red arrowhead performers so that they can track the position of this performer. In this case, the performers will receive the "cyan 5" message every two clock ticks. If the message needs to be sent every clock tick, this can be achieved by repeating the rule in a parallel instruction with the message sent to the other performers as in Figure 87.

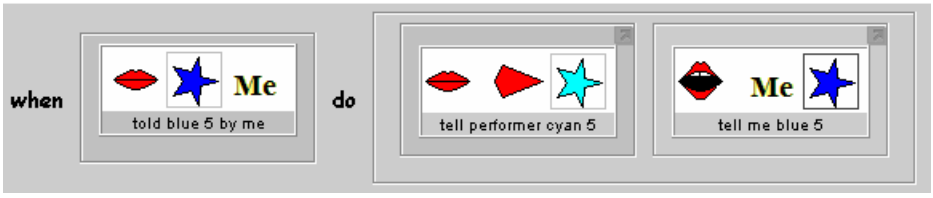


Figure 86 – A notification loop that repeats every two clock ticks.

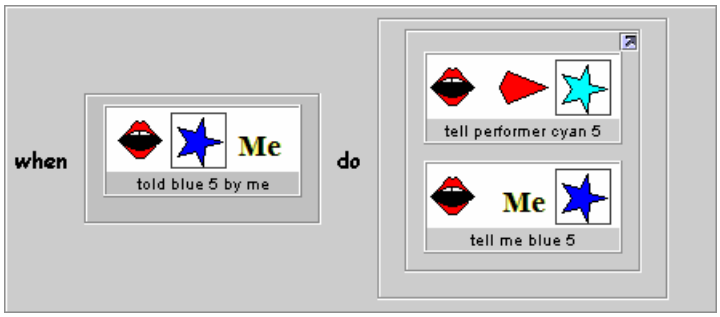


Figure 87 – A notification loop that repeats every clock tick.

These loops will never stop until the performer has been deleted from the world. If the user wants a loop to execute a set number of times, a variable must be used.

As an example, if the user wants a performer to morph backwards and forwards between two shapes 100 times, he or she could use the instructions shown in Figure 88; these rules are associated with a variable called "countdown". To start the loop the "countdown" variable is assigned the value 100. The value assigned to the variable is the number of times the loop will repeat. The loop stops when "countdown" is assigned the value zero.

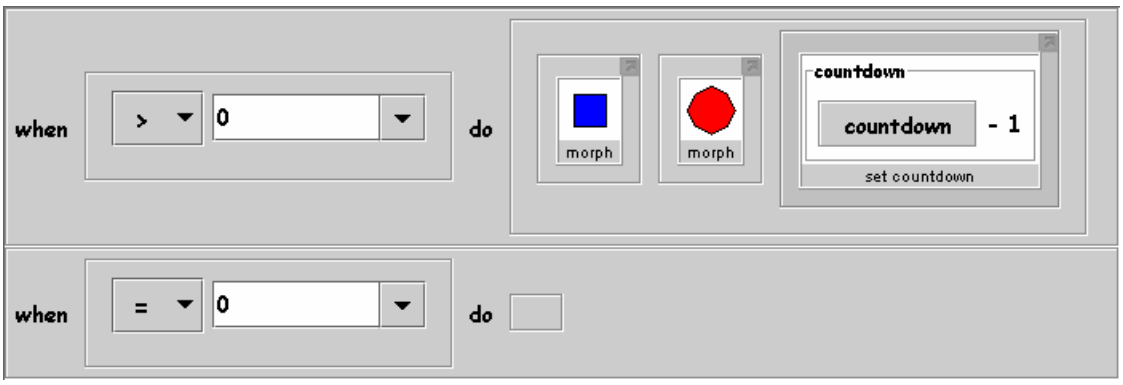


Figure 88 – A counted loop.

Starting and stopping repeat behaviour

Variable controlled loops can also be used to turn behaviour on and off. Only one variable rule can be active at any time for each variable. This means that each variable only adds one instruction stream to the list of instruction streams executing for each performer. When a new value is assigned to a variable, the current variable instruction stream is stopped immediately and the new one starts, even if the rule corresponding to the new value contains no instructions.

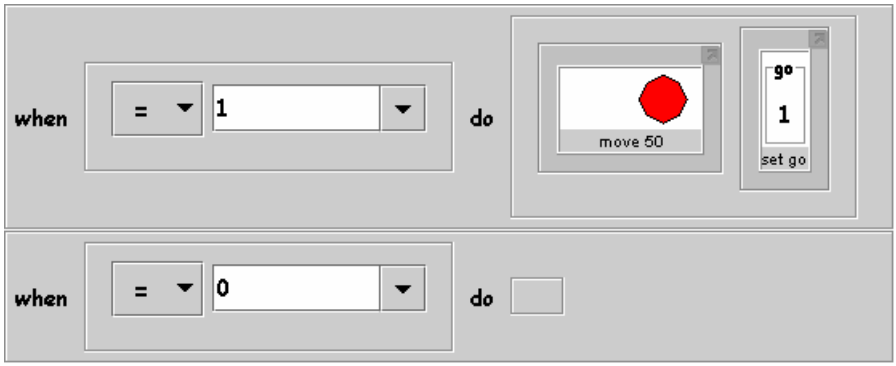


Figure 89 – Rules to control repeated movement.

Figure 89 shows two rules for a variable named "go". Assigning the value 1 to the "go" variable causes the performer to move repeatedly; the last instruction reassigns one to the variable. Assigning 0 to the variable stops the behaviour immediately.

Timed delay

Another useful pattern is the timed delay pattern. This can be used to slow down the results of key press rules if the user holds the key down for example. This is useful in a space invaders type game where you do not want repeating key presses to generate missiles until a certain time period has passed.

An infinite loop keeps track of the time the program has been running. Another variable keeps track of when the last missile was fired. When the user presses the fire key, the elapsed time since the last missile firing is stored in a variable called "delay". The "delay" variable has two rules. If the value being assigned is greater than the required delay a missile is fired, and the variable storing the time of

the last firing is updated. Otherwise, the "delay" variable is incremented. Eventually the value assigned to "delay" will be large enough and the missile will be fired. Figure 90 shows the rules of the "delay" variable.

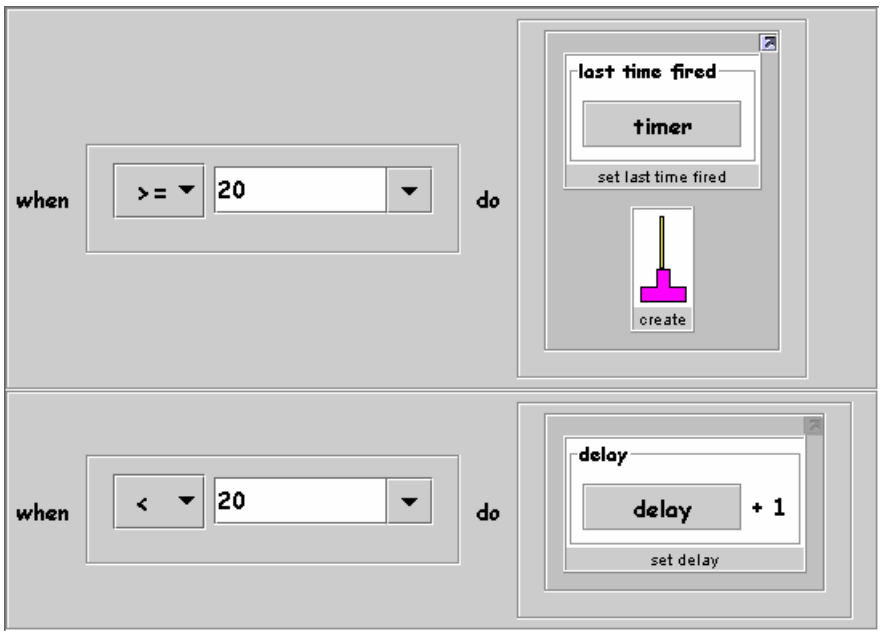


Figure 90 – Firing a missile with a delay of at least 20 clock ticks.

These patterns demonstrate how any combination of starting and stopping instruction streams can be implemented in Icicle.

Concurrency control in Icicle

With all of these streams of instructions executing for each performer, how does Icicle provide the type of control that locks, semaphores or monitors provide over parallel threads in an operating system?

Because a performer itself must carry out all state changes that occur on it, access to state variables is localized. So, while it is possible to have multiple interactions on a performer produce multiple instruction streams that conflict with each other, in reality (and in user testing), situations like this occur very rarely.

Each assignment to a variable happens reliably within one clock tick. This is true no matter how complicated the expression that evaluates to the assignment value. If several instructions assign values to the same variable in the same clock tick, all of the assignments are carried out as intended, but the last assignment will be the one that determines the rule to be executed by the variable in the next clock tick. Within parallel instructions, this is under the programmer's control because assignment instructions are executed from top to bottom in this case. In Figure 91 if the original value of the "balance" variable was 13, the value stored there after the parallel instruction would be 56, and the rule activated for the variable would be the one corresponding to storing 56.

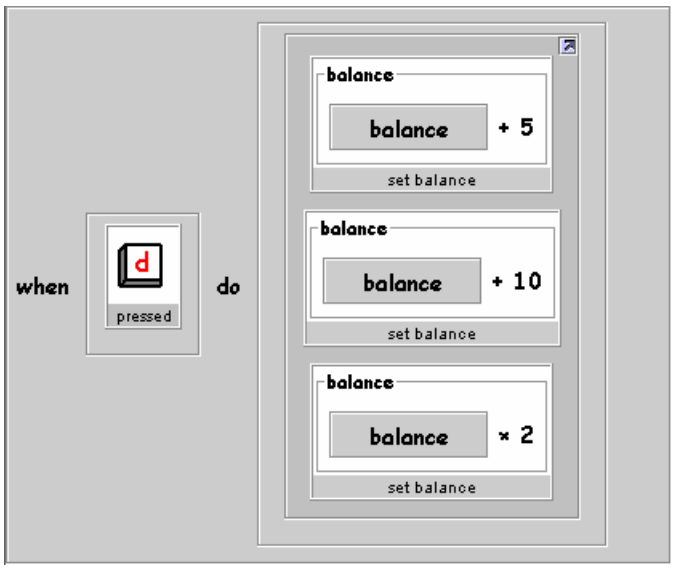


Figure 91 – Parallel assignment statements.

If the parallel assignment statements come from different instruction streams, the programmer cannot control the order of evaluation.

Complex conditions

Conditional statements are provided by the rules, but the conditions are simple, made up of one interaction. If a user wants a more complicated condition such as "when this performer collides with a star performer and the score variable is greater than 10", he or she must implement this using a test variable. When the required collision occurs, the score variable is assigned to the test variable. This variable has two rules, one executed if the value is greater than 10, and the other if the value is less then or equal to ten.

Every additional conjunction of interactions in a condition requires an additional test variable. This is cumbersome and a better technique for combining interactions should be added in the next version of Icicle. AgentSheets adds extra conditions into the "if" part of the same rule. In user testing these complex conditions were seldom needed.

Chapter 7

CHILDREN USING ICICLE

It could, like, expand your imagination.

A nine-year-old tester of Icicle.

There were several things I wanted to find out about Icicle when used by children, principally, whether children could use Icicle to produce the types of program they wanted, whether they would enjoy playing or working with Icicle, and if they would use Icicle in novel ways. Answers to these questions were derived by examining the programs children produced and by conducting a discussion session with the children. There were also specific questions about some of the features of Icicle:

- Could the children read and understand Icicle rules?
- Did the animation of instructions help the children understand the instructions?
- Could the children use parallel instructions to solve problems?
- Could the children use message passing and variables to solve problems?

In order to conduct tests to answer these questions, a group of children had to be taught how to use Icicle. During the time spent teaching the children, I encouraged them to make comments on the way things worked in Icicle and to make recommendations to improve Icicle.

CONDUCTING THE STUDY

The study was broken up into three sections. The first section was teaching the children how to use Icicle. The second section was a more formal evaluation of specific components in Icicle, and the third section was letting the children make their own programs with Icicle.

The school involved was the one in which the study on children's understandings of computer programs and programmers had been conducted. Twelve children were chosen for the study. They were selected by the teachers from across the senior classes of the school, four children from each class, six boys and six girls, aged nine or ten. The teachers were asked to recommend pupils who were creative and liked trying out new things. I hoped that creative children might use Icicle in interesting ways. I emphasized to the teachers, that I did not want computer experts, but that if children were good at working with computers that should not be a barrier to their participation.

I had to bring a laptop to the school with Icicle on it, as at the time, the school's computers did not have a recent enough version of Java installed on them to enable Icicle to run. This meant that I was constrained to working with the children in pairs, sharing the laptop. For a short period later in the study, I brought two laptops to the school for the children to work individually.

The children selected their own partners, giving two pairs of girls, two pairs of boys and two mixed gender pairs.

Timetable

I went to the school two afternoons a week for ten weeks, with a wrap up discussion session the following week. Each afternoon I would work with three of the pairs. Each pair saw me once a week and each session was approximately twenty-five minutes long.

There were six sessions devoted to working through the Icicle tutorial (see Appendix A). The tutorial was designed to cover all of the capabilities of Icicle. These sessions were very practical, with the children taking the ideas described in the tutorial and producing snippets of programs using those ideas. The work of the children was observed and notes were taken describing things they found difficult or behaviours they performed repeatedly.

These sessions were followed by two evaluation sessions that examined the specific features of Icicle listed earlier.

The children then had two sessions to develop their own programs from scratch. For these sessions, each child had a computer to work on. The children were encouraged to think about the design of their programs in the weeks before these sessions. In particular, they were asked to make drawings of what they would like their programs to look like and do.

For the final evaluation session, all children involved in the study were brought together and given the opportunity to demonstrate their programs to each other. A general discussion was held about Icicle; what the children thought about it, what its good points were and how it could be improved.

Difficulties and changes

Over the period of the study, there were some organisational difficulties. The study was conducted over the winter months and it was not uncommon for one of a pair of children to be unavailable due to illness, or an interschool cultural or sports activity. In this case, I either worked with one child or included the lone child with one of the other groups.

The version of Icicle used in the study still had some bugs. Fortunately, these seldom affected the programs the children produced. There were also problems with speed. On the laptop, some worlds crawled along when there were too many performers with multiple active threads. Because of these concerns, and other factors, changes were made to Icicle throughout the study. When the children found something that did not work properly, or as they expected, I endeavoured to correct this by the next session. Similarly, some of the requests the children made for additions to the system were satisfied. In this very limited sense the evaluation period also included aspects of participatory design (Druin, 1999; Muller & Kuhn, 1993; Read et al., 2002).

Logbooks

The children were given logbooks in which they were encouraged to make notes of things they found interesting, and to draw designs of performers and programs they were working on (see Figure 92 and Figure 97). The logbook page reproduced in Figure 92 shows the design of two performers, with keys along the bottom edge of the page to control the performers when the program is running.

Overall, the logbook approach was not as useful as intended because the children frequently forgot to bring them to school. The books could have been kept at the school but I wanted the children to take them home so that they could record ideas in them.

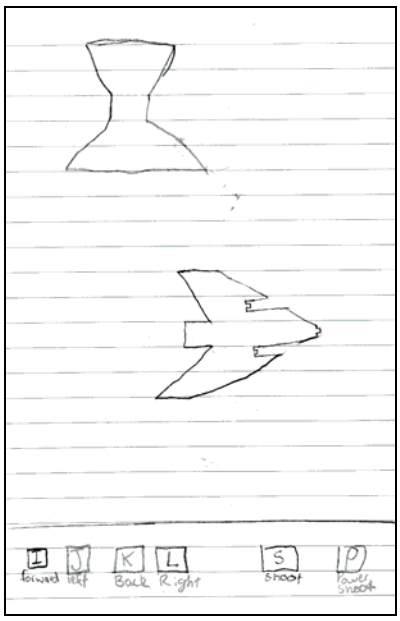


Figure 92 – A page from a child's logbook.

Order of presentation

The sessions will be covered in the order the children experienced them. First, I shall present general comments that were gathered from the sessions on learning Icicle. Then I shall cover the evaluations of the specific questions. This is followed by a discussion of the principal questions, which includes descriptions of some of the programs produced by the children and some quotes and suggestions from the children.

LEARNING ICICLE

The first six sessions with the children were to familiarise them with all components of Icicle. In order to structure this time I produced a tutorial booklet for each child that described one path through Icicle. This booklet is included as Appendix A. I did not seriously expect the children to read these booklets in between sessions, but one boy was so interested in Icicle and what he could do with it, that he always read ahead. He could tell me how to do things before he had been shown.

The tutorial material was presented in six sessions of twenty-five minutes and covered:

1. The production of performers and shapes, and using the morph panel.
2. Dealing with interactions, giving performers instructions and editing instructions.
3. Keyboard interactions and making parallel instructions.
4. Creating performers and the creation editor.
5. Message sending and turning to face other performers.
6. Variables and simple variable interactions.

Rather than make the children read the tutorial material by themselves, I explained sections of the tutorial and asked the children to create their own performers and carry out tasks similar to those described in the tutorial.

It was mentioned earlier, that small changes were made to Icicle during this period, when children made recommendations or had difficulties. Consequently, the version of Icicle described in Chapter 4, Chapter 5 and Chapter 6, is the modified version of Icicle and does not exactly match the version described in Appendix A. Most of the differences are described in Appendix B.

Complaints and misunderstandings

The main complaints the children made about capabilities missing from Icicle are described here, along with some misunderstandings the children had.

No circle shape

The children wanted a circular base shape to be added for performers. The octagon was viewed as not being circular enough. Even so, the octagon was used as the basis for many performers e.g., spaceships (Figure 93). More generally, the children wanted a larger set of basic shapes to start their performers.

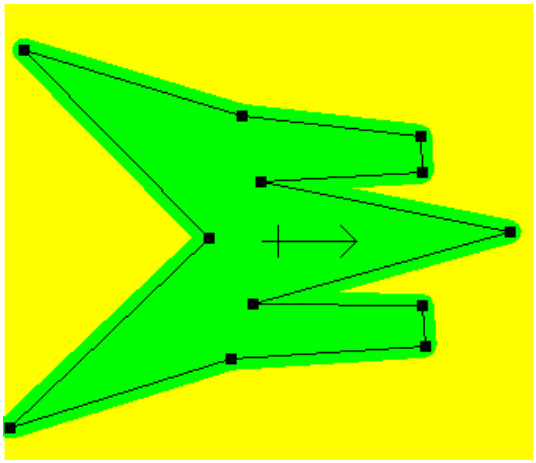


Figure 93 – A spaceship derived from the octagon.

Shapes or performers

Also associated with shapes, it was noticed throughout the study that some children tried to use the different shapes of a performer as though they were different types of performer. The problem occurred when a child had an open performer box and created a new shape, intending the shape to be a new performer, rather than using the "create new performer" tool. These could have been simple mistakes or they could indicate that the hierarchy of performer box (corresponding to the type), performer (as an instance of the type), and shape (as a property of a performer), was lost on some children.

Zoom parameter

The great failure of the Icicle instruction set was the zoom parameter. In an effort to avoid fractional values the parameter was expressed as a percentage, so "zoom 200" meant twice as big, and

"zoom 50" meant half as big. The children could deal with numbers above 100 easily enough, but had no idea of what numbers under 100 meant, and could not tell me what number was needed to make a performer a quarter of its original size. When the children were asked what numbers would best represent getting smaller, a common answer was to use negative numbers, e.g., -500 to mean a fifth of the size. The reason for this appears to be that the children were perfectly happy with negative values being used to undo moves and turns. In these situations the parameter is additive but for zooming the parameter is multiplicative. The children were over generalizing. Fortunately, as most zoom instructions were produced by example, the children did not need to type a value in for the parameter.

Sound

Although, in accordance with guideline 10, it was always intended to have sound instructions as part of the Icicle environment, this was not implemented in the version the children first used. After many requests from the children, usually when one performer collided with another, a sound instruction was added.

Discovering interesting things

Even during the period of learning Icicle, the children used it in interesting ways. Some of these are listed here.

Driving multiple performers

When driving a performer around the world with key presses, several children dropped multiple performers of the same type into the world and drove them all around simultaneously. Of course, depending on the orientation of each individual performer, the results of the moves were quite different. This effect was played with for some time and demonstrated the playfulness that working with Icicle evoked from most of the children. The liveness properties were possibly instrumental in this.

Parallel key press rules

Because of the way multiple interactions are handled in parallel for the same performer, you get the effect of parallel instructions by causing two or more interactions to occur at the same time. Children discovered this for themselves when driving a performer around the screen. If one key press made the performer turn to the right and another made it move forwards, by pressing both keys you made the performer curve to the right.

Parallel instructions

A discussion of the understanding of parallel instructions will occur later in the chapter, but an interesting feature became apparent during play with the parallel instructions. One of the boys starting using parallel instructions to turn performers through multiples of 30°, e.g., to turn through 90°, three turns of 30° were combined in one parallel instruction. This also had the effect of speeding the turn up, as each of the turns was truly being executed in parallel. He also used the same technique for moving performers very quickly across the world. This was a completely unexpected but clever use of the parallelism mechanism.

Interesting effects

Interesting effects were discovered by several children when using create rules, e.g., creating a performer of the same type as the creator and deleting the creator produces the effect of the performer jumping across the screen leaving a shadow of itself behind.

UNDERSTANDING ICICLE

I wanted to test several areas of Icicle. These tests were conducted after I had worked through the tutorial with the children.

Reading rules

As was explained in Chapter 4, one of the problems with PBD systems is that most of them do not produce source code that is readable by the user. If children could inspect Icicle rules and describe the effects of the rules, that would demonstrate that Icicle rules were *readable*.

Part of the difference between Icicle and ICE was the animation of instructions. If children could describe animated instructions more correctly than non-animated instructions, that would demonstrate that animation was useful.

To answer these questions I presented the children with performer rules in two forms: paper printouts and animated rule lists in Icicle. The instructions in both cases were equivalent and I varied the order of presentation. The children were asked to describe when the rule would work, what the instructions would do, and, in some cases, to explain what path the performer would move through on the screen as a result of carrying out the instructions. Appendix C shows the paper printouts.

Understanding the conditions

The first part of understanding an Icicle rule is understanding when the rule will run. This happens when the condition is true.

Half of the children described the "when nothing" rules as running when the performer was doing nothing else or was waiting for something else to happen. The other half said this ran when the world was started or the performer was dropped into the world. The first explanation is the correct one but the second is true in most situations.

One of the questions was based on the instruction shown in Figure 94. The children were asked what shape the performer would move through on the screen if nothing else were happening to it. One child had no idea; one thought it might move like a staircase, the rest got it right, even those whose descriptions of the condition were not completely correct. This indicated that they understood, not only the instructions, but also the fact that the rule would repeat if nothing else were happening.

Figure 94 – Test instruction.

The other conditions tested in the samples included key press interactions and collisions. All children explained these conditions correctly.

There was no difference in their understandings between the paper versions and the animated versions of any of the rule conditions.

Understanding the instructions

The second part of understanding an Icicle rule is understanding what the instructions mean – what effect they have. Most instructions were tested.

Moves and turns

Almost all of the children described moving by a negative amount, as moving to the left. This was because the animations showed moving forwards, as going to the right and moving backwards, as going to the left. Interestingly, the children described the paper version as moving to the left as well. In the pictures produced for the paper version, the performer was on the left hand end of the instruction box for negative moves and at the right hand end for positive moves.

Turn instructions were understood much better from the animated versions than from the paper versions. In order to eliminate misunderstandings of their descriptions, I asked the children to show me with their hands the way they thought the performers would turn. Almost half the children misunderstood a "turn -40" instruction in the paper version but they all described it correctly from the animated version. This was a real test of whether they knew what a negative turn was because the performer was a square and the static representation of the turn on paper could have been interpreted as turning clockwise or anticlockwise. The animation helped in this case.

Morphs and zooms

The paper versions of the morph instructions showed the result shape. In this case, the paper version of the instruction was understood by the children just as well as the animated version. Having the label "morph" under the picture made this possible, whereas I believe the animated version would have been understandable even without the label.

One child got the direction of a zoom instruction wrong in the paper version. The illustration showed the final size, which was smaller, but because the instruction was labelled "zoom 25" the child thought the performer got bigger. The child described the zoom correctly from the animated version.

From the animated version, another child realised the two zooms, "zoom 25" and "zoom 400", undid each other, once again demonstrating the usefulness of the animated instructions.

Deletions and creations

The delete instruction was understood equally well from the paper version and the animated version, but the create instruction was not. The illustration for the create instruction showed two performers, the creator and the created. Even though it was possible from the context of the instruction, to say which performer was being created, one child thought the paper version showed the creation of two different performers. He understood the animated version correctly.

Turning towards

After a collision interaction, one of the instructions was a "turn towards" the other performer. The children were shown two versions of this, with two different types of performer involved in the collision, an octagon and a square, with the octagon turning towards the square, and a version with two squares, one square turning towards the other one. The instruction with the two different types of performer was understood correctly, but there was some confusion with the two squares. Many of the children did not know which square was doing the turning. Of course, if both squares were of the same type when they collided in the world, they would carry out the same instructions and so both squares would turn towards each other. This was just as confusing with the animated version as it was in the paper version.

Understanding parallel instructions

The children were presented with some parallel instructions. In the paper versions, the parallel boxes were expanded to show the individual instructions; in the animated version, the parallel boxes were collapsed showing the resulting parallel instructions. The reason the paper versions were expanded was that a static picture of a performer with the label "parallel" as its description would have been impossible to decipher – there is not enough information in the picture.

Interpreting the paper version was difficult, not only did the children have to remember that instructions stacked on top of each other were performed in parallel, but they also had to try to work out what happens when one type of instruction happens at the same time as another type of instruction. Not surprisingly, most children could not correctly explain what a parallel instruction would do from the paper version. Almost all of them could do this easily from the animated version. The only incorrect answers came from two children who described an animated instruction with a morph and a move occurring together, as a morph. They did not seem to realise the performer was moving as it morphed. This may have been because the morph animation was more dramatic than the simultaneous movement.

Conclusions

- When presented with animated Icicle rules, the children could describe the conditions and instructions correctly. This indicates that Icicle rules are *readable*.
- In many cases, the paper representations were adequate for understanding the instructions but the animated representations helped the children correctly understand the instructions.

The animations of instructions were appreciated so much by the children that they recommended that the Icicle world controls down the left-hand side of the main window should be more like them. Currently, these are just buttons that display a different icon when the pointer floats over them.

Writing parallel instructions

Most of the children could successfully interpret parallel instructions from the animations. I also wanted to determine how easy it was for them to use parallel instructions to achieve particular results. For this test, the children were shown some existing performers and then asked to give them behaviours that required parallelism.

The first task was reasonably straightforward: they had to make a bird flap its wings as it flew across the world. All the children realized they would have to use morph instructions with move instructions and most of them got the combination right. One child dropped all four instructions in one parallel box, this had the effect of the two morph instructions operating in parallel and cancelling each other out. Another child could not work out how to get the bird to go back to its starting shape – there has to be a second morph if you want to see the first one again.

The second task was much more difficult. They were asked to make Superman fly in a figure eight. All of the children realised they had to combine moves with turns but none of them could make Superman fly in a figure eight in the time they had. Some got close, with Superman flying in interesting curves, but it was a conceptually difficult task. They found it much easier to make Superman fly in a circle.

These results matched observations made during the sessions on learning Icicle. The children did not know beforehand what dragging two or more instructions together into a parallel instruction would produce. It was however easy to see from the instruction animation and if the effect was not as required the parallel instruction could be expanded and altered.

Conclusions

- Children had no difficulty producing parallel instructions in Icicle.
- Children did have a difficulty producing parallel instructions to create specific patterns of movement.

Implication

The test to determine whether children could produce specific parallel instructions used turns and moves to generate curved motion. Understanding how to produce a specific curve was very difficult for the children. A technique to produce curves by demonstration, similar to the *squiggly* mode of ICE, may be helpful in these situations.

Understanding message passing and the rules associated with variables

Message passing and variables are the most abstract topics dealt with when programming in Icicle. It was expected that the children would not find them easy to work with.

The children were presented with scenarios and asked to complete the programs to make them achieve particular results. Except for two children, whose partners were absent, the children worked in pairs to solve these puzzles.

The first scenario was of a set of traffic lights, with only two lights. By pressing the down arrow key, the red light was to go on and the green light was to go off. The up arrow key was to do the reverse. The expected solution was to use a key press interaction to change the colour of one performer and then send a message to the other performer to change its colour[1]. Three of the pairs

[1] Why not send the same key press interaction to the two performers? With the current implementation, each key press interaction only goes to one type of performer because a performer (and hence type) has to be selected as the destination of the interaction before the rule is defined.

solved the puzzle without assistance. The fourth pair and the two children working alone needed small amounts of prompting but could complete the puzzle when started.

The second scenario added a car to the world with the traffic lights. The car was supposed to move when the green light was showing and stop when the red light was showing (Figure 95).

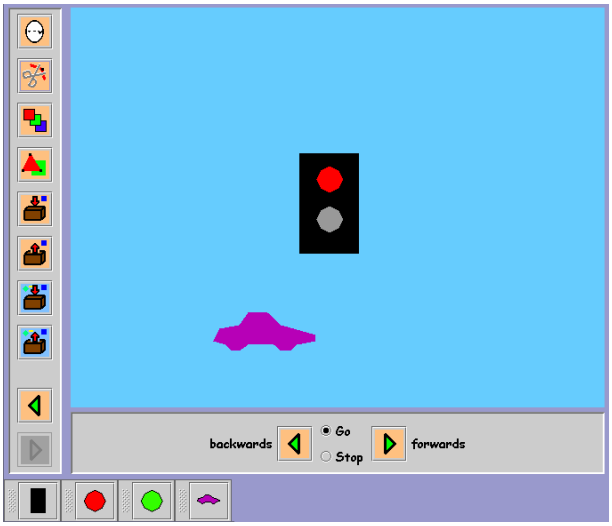


Figure 95 – The traffic light test.

This was a more difficult problem. To make the car move continuously while the light was green and yet stop moving when the light was red, required a variable in the car to control its movement. The car already had a variable called "go", and a rule that specified that when "go" was assigned a value greater than zero it would move the car and then assign itself the value one – moving the car continuously. One solution used the red light to send a message to the car, and when the car received the message, it would set its "go" variable to zero. More than half of the groups solved this puzzle.

The third scenario had two performers, a steering wheel and a car (Figure 96). The task was to make the car turn to face the same direction as the heading of the steering wheel performer. This required a message to be sent from the steering wheel every time its heading changed. The heading changed as the user dragged the heading arrow of the performer. When the car received the message it had to set its own heading variable to the value of the steering wheel's heading. The children realised the steering wheel had to send a message to the car but only two children, out of the ten, understood how to get the steering wheel's heading – it is only visible in the variable assignment expression editor.

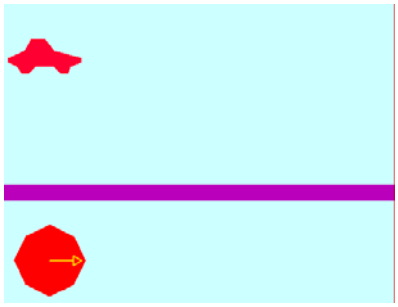


Figure 96 – The steering wheel test.

The last scenario was to make a three-light traffic-light work. When the user pressed the down arrow key, the green light was to go off and the amber light was to go on. After a few seconds, without extra key presses, the amber light was to go off and the red light was to go on. The up arrow was supposed to make the green light go on and the red light go off as before. The expected solution to this required a counter variable to turn the amber light off after a certain period of time. Only one group reached this puzzle during their twenty-five-minute session and they got it right by making the amber light morph several times to the on colour before sending a message to the red light. This had the required effect of causing a delay while the amber light was on. The technique was not noticeable to the user because morphing from amber to amber caused no change to the performer.

Conclusions

- The children could use message passing to solve problems if they were reminded that message passing instructions existed.
- Most children found working with variables difficult.

In the last session when the children were asked what things they found most difficult to understand in Icicle, the almost unanimous topic was variables. This difficulty with variables is in agreement with other studies (du Boulay, 1989).

Implications

The message passing instructions were not used frequently by the children. Because of this, several children did not think of using them. If the message passing system was made easier to understand, the children might use it more often. Dragging a message from one performer to another could be used to demonstrate sending a message. Whether this would be an improvement, would have to be determined by testing.

It may be possible to provide extra assistance to do the most common tasks that require variables. Even adding one to a variable requires using the expression editor at the moment. Similarly, as was mentioned in Chapter 6, a simple interface could be provided to rules that repeat until they are switched off. This function is currently provided by variable rules but could be presented in a more straightforward manner.

PRINCIPAL QUESTIONS

- Could children use Icicle to produce the types of program they wanted?
- Did they enjoy playing or working with Icicle?
- Did they use Icicle in novel ways?

To answer these questions the children were given time to produce a program of their own. This was followed by a group demonstration session and discussion about Icicle. The children had two twenty-five minute sessions to produce programs of their own designs. They all wanted to make games. I asked them to draw on paper what they wanted their games to look like, and to think about the movements of the performers and the ways of scoring. They produced a range of games, from sports simulations to arcade style games of skill.

Games produced

I will briefly describe three of them. In Figure 97 you can see the design sketches the children made and the final Icicle worlds produced from them.

Watermelons

The game shown at the top of Figure 97 was very simple in principle but involved a very clever effect. The girl in the game can be moved backwards and forwards using key presses. The clouds move across the sky and watermelons fall from them. The idea is for the girl to catch the watermelons before they hit the ground. What made this game particularly interesting was the composition of the girl. She was made up of four performers.

Several children had requested some way of producing multicoloured performers and I had told them that in the current version of Icicle this could not really be done. Helen[2], the nine-year-old girl who produced this game, started constructing a face performer and then made hair, skirt and legs performers. The different performers could easily be placed in the correct orientation, size and position, but when Helen came to move the girl object, the question was how could all of the performers be moved together? Almost immediately, she said, "Messages!" She realised that if one performer as part of the girl responded to the move key presses, that performer could send messages to the other performers comprising the girl, and they could move in unison. Helen chose the skirt to respond to the key presses and sent messages in parallel to all of the other parts of the girl when the skirt moved. The only downside to this solution was that there was a one tick delay between the skirt moving and other three performers moving. Helen did not mind. She said that she liked the effect.

Soccer

The middle game in Figure 97 shows one girl's novel version of soccer. The players were represented by different coloured star-shaped performers. There were spinning performers that the ball bounced off, in order to make the game more challenging, but the most interesting aspects were the goals. Each goal consisted of several performers. The idea was that different coloured sections of the goals were worth different scores. The game was intended to be a two-player game but after the fifty minutes of work, only the controls for one player had been completed.

Shoot

The bottom game in Figure 97 was produced by a ten-year-old boy, Leo. The arrow on the right-hand side could be moved up and down by the player. This arrow fired sticks across the world to the left. The object of the game was to hit the performers on the left-hand side of the world, avoiding the other performers on the way. There were four lines of performers continually moving up and down the screen. Hitting different performers increased or decreased the score variable.

[2] All names of children in the study have been changed.

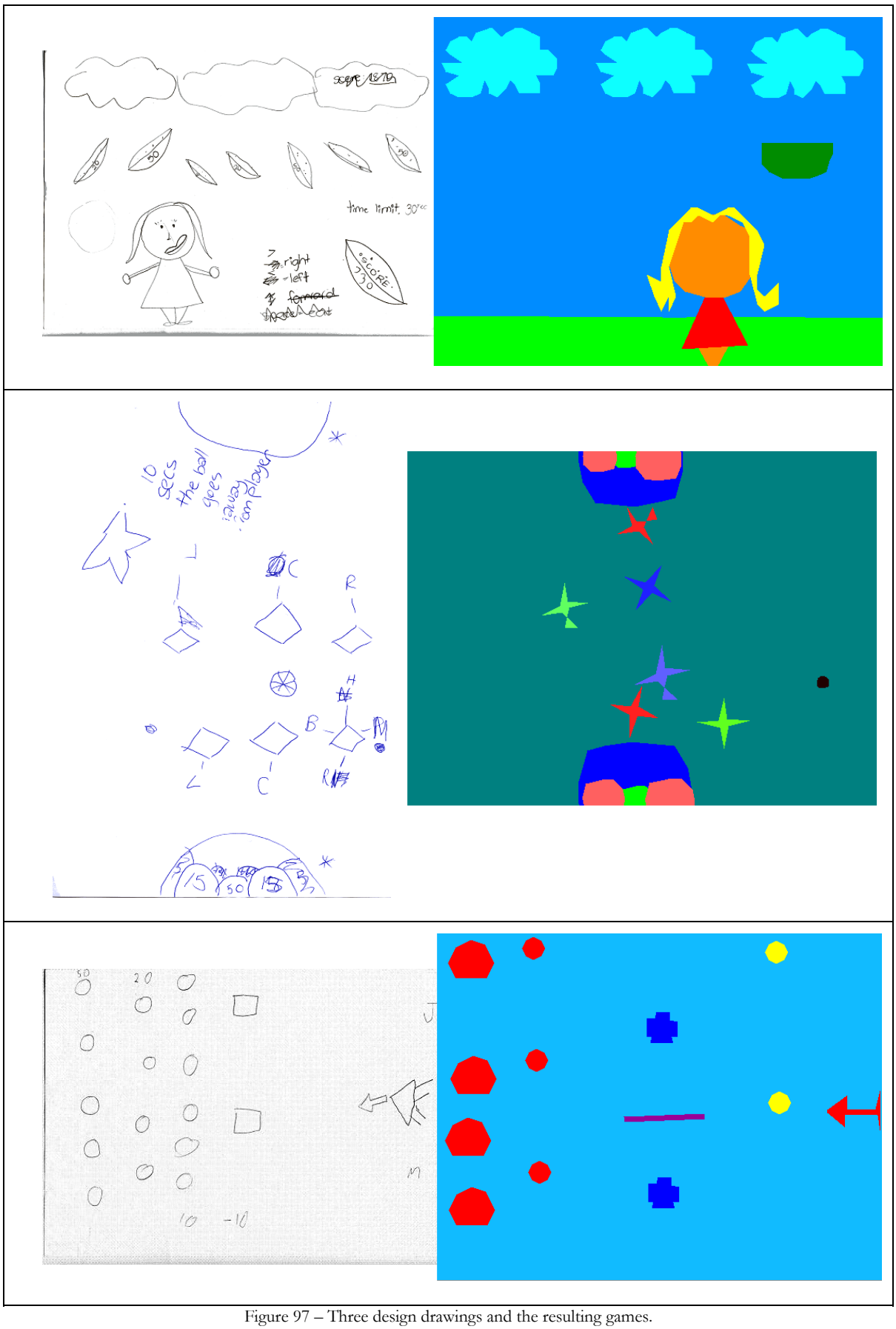


Figure 97 – Three design drawings and the resulting games.

The arrow on the right had to move up and down the screen, but because Leo had created the shape pointing to the left, it was not trivial to make that happen. A move instruction moved the arrow left and right rather than up and down. The simplest solution would have been for Leo to rotate the shape in the shape editor. Instead, he used a parallel instruction with a move command and a command to set the "heading" variable (Figure 98). Because of the instruction to change the performer's heading to point straight up the screen, the performer will actually move upwards. The direction the performer is facing is not altered by the "heading" variable.

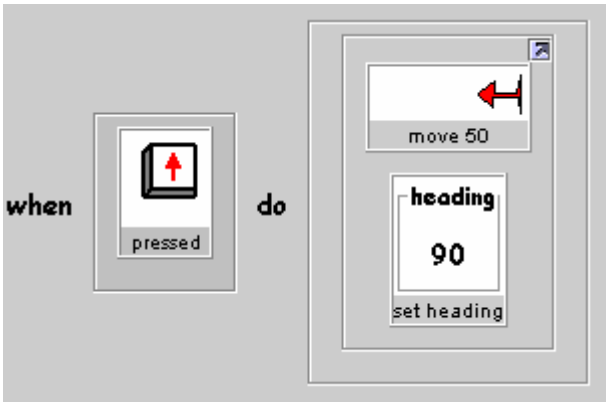


Figure 98 – Moving the arrow up the screen.

The use of variables in the programs

Several of the children used variables in their programs, usually to add or subtract values from a "score" variable. They usually needed assistance with this. There should be a way to "add 1 to score", rather than requiring "score + 1" to be inside the set variable instruction.

Helen successfully used a variable to drop watermelons at regular intervals from the clouds in her program as well as adding to a score variable if a watermelon hit the girl's hair and subtracting from the score variable if a watermelon hit the ground. One boy had variables for lives, score, number of missiles and even a complicated condition – a missile would be fired when the player pressed the fire key only if the number of missiles was non-zero.

Programs that did not work

Not all of the programs the children produced worked as intended. One program in particular caused me to wonder what fault in Icicle was causing its strange behaviour. It turned out that it was not a fault in Icicle but rather an unusual sequence of interactions.

The game was produced by Helen's partner, Edna[3]. Edna produced a game with a dog that moved backwards and forwards, and bones that fell from the sky. The problem was that very quickly dozens of bones were falling from the sky and the program got very slow. When a bone hit the ground it created a new bone further up the screen. Edna's intention was for there to be only one bone on the screen at a time. What had gone wrong?

[3] With many of the pairs who worked together, both children produced programs of a similar type.

The problem was not with the rules of the performers, it was in the way Edna had constructed the ground. The ground was represented by large green rectangular performers. In one section of the world two of these performers overlapped. This meant that when a bone hit this part of the ground there were two collisions with two separate performers and the one bone produced two. From then on the number of bones grew exponentially. If the ground had been constructed from one performer this problem would not have occurred but it was very difficult for Edna to understand what was going on.

This problem demonstrates the lack of debugging facilities in Icicle. The best that can be done to solve a particular problem is to step forwards and backwards through the program, examining the results. Variables can also be inspected and modified between steps. In Stagecast Creator, each rule that can potentially execute has a green light that shows up next to it. Something similar could be done in Icicle.

Could children use Icicle to produce the types of program they wanted?

It looked as though the children were making the games they wanted to make. The results certainly matched the drawings and descriptions the children had produced for their games before they started (Figure 97).

This was reinforced in the group discussion when different children said:

> *You can practically make anything you want and do anything you want.*
>
> *You get to add lots of things. You get to make your own things.*
>
> *Make whatever you want.*

When queried further, the children really did believe that Icicle would allow them to make whatever they wanted. They had not realised that text was missing from Icicle. When I pointed this out to them, they were surprised that they had not thought of that.

Conclusion

The children believed that Icicle allowed them to make the programs they wanted.

Did they enjoy playing or working with Icicle?

There were children who did not enjoy working with Icicle. One pair, of two girls, lost interest early on and wanted to use the time to play around. Even though I covered most of the material in the tutorial booklet with them, they never got beyond creating and colouring shapes, dropping them in the world and giving the performers random behaviours. In the final evaluation session when I gave the children the opportunity to demonstrate their programs to the other children, these two were the only ones who did not want to.

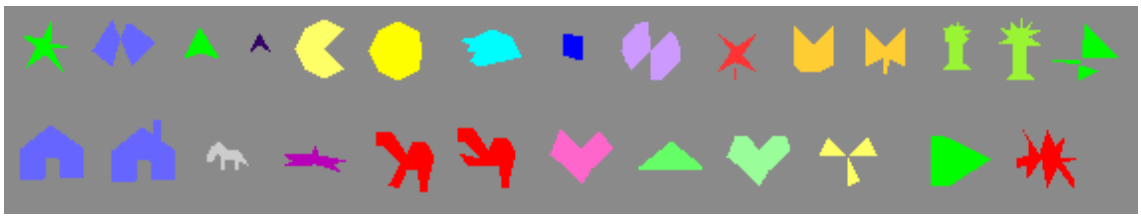


Figure 99 – Shapes children made in the first session.

They were the exceptions. From the first session (Figure 99), when they constructed shapes and morphed them from one to the other, until the last session, working on their games, the other children remained highly enthusiastic. They responded with these comments to the question, "What was the best thing you liked about Icicle?"

> *It could, like, expand your imagination.*
>
> *You can let your imagination go wild.*
>
> *I could use a computer for once.*
>
> *It's easy to use.*

I found it particularly gratifying that the children associated Icicle with imagination, as one of the worries about computer use by children, mentioned in the Introduction, was the stunting of imaginative thinking.

Conclusion

Most children enjoyed using Icicle.

Did they use Icicle in novel ways?

Even in the short time they had using Icicle, less than three-and-a-half hours each (and most of that time sharing a computer), the children did things I had not imagined with Icicle. Two of them were moving multiple performers in unison, and making quick moves and turns using parallelism.

The shapes produced and the behaviours of some performers were also frequently original – in fact, quite strange.

Conclusion

The children did use Icicle in novel ways.

FURTHER COMMENTS AND RECOMMENDATIONS FROM THE CHILDREN

In the final discussion session, the children made many other comments. I include the main recommendations here.

They requested keyboard shortcuts for some actions, in particular, creating a new performer and copying a performer. One child suggested giving the performers single letter names, and making a new performer appear in the world when its letter was typed. This can be done already, using one performer as the creator of other performers, when their letters are typed.

Copying rules from one performer to another was one suggestion. The ability to select an Icicle file by inspecting pictures of the world running was another. The children believed this would help younger children, who could not read, to use Icicle. They also said that children who could not read would have difficulty changing distance and speed parameters.

The children said that there should be a program instruction to move to a new world (or level). One girl suggested that I could have a special performer and when it moved off the world it moved to a new one and the program did too.

Being able to use Icicle worlds over a network was suggested for multi-player games.

Consistent with guideline 10, they suggested the ability to record their own voices and produce their own sounds for the sound instruction. Similarly, they requested the ability to import scanned pictures of themselves and other objects to be used as performers and backgrounds.

The boy who said that Icicle expanded your imagination was the child, who throughout the study, planned things most carefully before trying them. He suggested that there should be a list of all possible rules for a performer when it was first constructed, so that he did not have to wait for the situation to arise before he could describe the instructions for that event. A way of turning this on could be provided to make Icicle more convenient for the planners. At the moment, it can be argued that it is more convenient for the bricoleurs but even the planners enjoyed working with it.

WALK-UP TESTING

Since the original user tests were conducted, Icicle has been demonstrated at conferences and open days. In these situations, children have been walking up to computers running Icicle and been given very brief demonstrations of how to use the system. I have been surprised, but very pleased, at how quickly the children have picked up the ideas and produced performers and rules. Getting children off Icicle at the end of the day has been the most difficult task.

THE ACHIEVEMENTS AND LIMITATIONS OF ICICLE

Now that Icicle has been fully presented, and its use by children described, it is possible to compare it with the guidelines from which it was developed. Unsurprisingly, it matches most of them quite well. However, there are many types of programs that children might want to develop that cannot be produced by Icicle; some of these will be considered here, along with techniques to work around these limitations and produce similar types of program with the existing version of Icicle. To make some of these programs easier to produce in Icicle, several additions and changes are also recommended.

The question of whether Icicle is equivalent to a Universal Turing Machine (D. I. A. Cohen, 1986) is also relevant, because the answer shows the theoretical power of Icicle.

HOW DOES ICICLE MEET THE GUIDELINES?

1. *A programming environment for children should be able to produce programs of a wide variety of types, especially the types that children most want to produce.*

This was answered affirmatively by the children who used Icicle in Chapter 7. Some limitations of Icicle are discussed in the next section, but Icicle is flexible enough to implement a wide variety of programs. Complicated programs do require complicated facilities, such as the use of variables and message passing, but lots of programs can be produced in Icicle without resorting to these techniques.

2. *A programming environment for children should give the children objects they can see and manipulate to program with.*

Both performers, and to a lesser extent variables, are objects that can be seen and manipulated. Icicle continues in the tradition of the Logo turtle in making the objects of computation concrete.

3. *A programming environment for children should allow the objects to be manipulated in a series of small reversible steps.*

Icicle rules are constructed either by demonstration or by directly selecting instructions. Each of the steps taken by the user when producing the instruction list for a rule can be undone and redone. When defining a rule at the request of the system, any changes made are automatically undone and done on the performer in the world the rule is being produced for.

Each instruction in Icicle is conceptually small in that it is easy to understand. They may take a long time to execute, e.g., moving 1000 steps, but in these cases the instructions will be carried out as a long sequence of small steps, as was described in Chapter 6. These small steps can be run forwards and backwards in the Icicle world using the forwards and backwards buttons.

4. *A programming environment for children should make the sequences of instructions in a program explicit and easy to follow.*

The rules in Icicle are represented by animations showing the effects of the instructions. To convey the order of execution, the interaction and instruction animations are activated in sequence. A child can observe a mini movie of what each rule does by following the sequence of animations.

5. *A programming environment for children should be interactive or lively.*

The performers in Icicle use several techniques to appear lively. They can move and respond to events. They maintain their own state and behaviours. Any changes made to the performer type are automatically reflected in all matching performers in the world.

The Icicle environment is modeless, with all of the tools available even when the world is running, adding to its interactive and lively nature.

6. *A programming environment for children should provide a variety of ways of using the environment, starting with a very simple entry level and a gentle learning curve.*

In Chapter 7 we saw how easy children found Icicle to use. It does have a simple entry point – make a shape and drop it in the world. It is then trivial to make the shape move around in the world – drag the shape when asked for a rule. Because the system queries the user to provide rules as required, the users are given assistance as to what they should be doing next.

To do more sophisticated things, users need to use the various instruction editors; these are hidden from the new user because they only appear when instructions are right-clicked on. Even more difficult is the concept and control of variables and their rules. These provide the most power and enable the production of complicated programs.

7. *A programming environment for children should make its models explicit.*

The computing model for Icicle is simple – performers do things when interactions occur. This is made explicit because the user has been asked to define rules when the interactions happened the first time. With most programs there is no need to work out which rule should be active when multiple interactions occur – all interactions will work.

8. *A programming environment for children should be fun.*

Another question answered positively in Chapter 7. Icicle tries to maintain enjoyment by providing large effect for little effort. It also aims to reduce frustration by making it impossible to produce a syntax error. Many of the other features such as liveness and the animation of instructions contribute to making Icicle enjoyable.

9. *A programming environment for children should not be ready-made.*

Apart from the four standard shapes and a large number of sounds, everything in Icicle must be constructed by the user. Even the four standard shapes were chosen to encourage children to modify them to look like something else.

10. *A programming environment for children should include the ability to produce and collect resources such as pictures, animations and sound.*

Icicle does not do this very well. It allows the simple production and modification of shapes for the performers. It also allows a user to load in performers from other Icicle worlds, but it provides no way of producing or collecting new sounds. Animations are of course produced by performers moving and changing in the world. They can only be shared in the sense of loading them from other worlds.

11. *A programming environment for children should provide high-level instructions that correspond to the things children want to represent in their programs.*

Icicle provides a set of instructions and interactions that allow objects to move around the screen and interact with each other and with the user. It is interesting, that even though the instructions are simple and may not be regarded as 'high-level', they are very powerful due to the parameters that can be set either by demonstration or by the instruction editors. In particular the *create performer* instruction is simple to understand but can produce sophisticated effects.

In Icicle, the interactions should also be considered part of the instruction set. They are conditions – and in the case of the collision interactions very high-level conditions – that do exactly what is needed for many games and simulations.

12. *A programming environment for children should be usable by children when they are most interested in making their own programs.*

Can Icicle be used by children around the age of eight? In Chapter 7 we saw that nine and ten-year-olds successfully used most of the features of Icicle. Some of the features are difficult to use and some recommendations to make Icicle better are described at the end of this chapter.

CAPABILITIES ICICLE DOES NOT HAVE

The first guideline for Icicle was that it be able to produce programs of a wide variety of types. In Chapter 7, the children who used Icicle believed it produced the types of programs they wanted, but what limitations does Icicle really have?

Icicle can be used to produce games and simulations of many types (see Appendix D), relatively easily, but there are some programs that cannot be implemented with Icicle. What types of program are possible, but difficult, or unrealistic to implement with Icicle?

You cannot do some things with Icicle that you can do with most other programming languages. Icicle does not have any way to send output to devices other than the screen and MIDI (Hunt & Kirk, 1999) sound processors. This means that programs cannot print, they cannot communicate with other programs locally or over the internet, they cannot work with files and they cannot control external devices. The abilities to print, communicate over networks, and modify files are useful for any programming environment and should be considered seriously for future versions of Icicle.

The problem of not being able to print from within Icicle can be alleviated a little by the operating system. The contents of Icicle windows can be captured and sent to printers under all common operating systems.

The problem of not being able to communicate over a network does not have such a partial solution. The two subsidiary guidelines derived from the study in Chapter 3, were concerned with the sharing of programs and being able to work as a group. Network or Web access makes this easier. As Icicle is written in pure Java it is a simple process to make it available as a Java applet (Gosling, Yellin, & The Java Team, 1996) in a Web page. This does not give it any communication abilities but a Web site can easily be produced to store Icicle programs and performers. A repository of performers could even be provided, so that performers could be added to an Icicle applet by dragging.

The problem of not being able to work with files is only true from within Icicle programs. Obviously, Icicle worlds and performers can be saved to and restored from files at the user interface level. This means such things as storing top scores can be done by saving the current world to a file.

Icicle does not provide access to libraries of code written in other languages. This means that standard libraries that handle complex algorithms or deal with complicated data such as movie or sound files cannot be used. As we shall see shortly this does not mean that we cannot write such libraries in Icicle itself, but it would be exceedingly tedious to do so and impossible in any realistic way.

Several types of program appear to be impossible to implement in Icicle. Some of these are the types of program that children might realistically want to produce:

- Programs that use three-dimensional graphics displays.
- Programs that display text.
- Programs that use particular data types or collections of data. Icicle has a small number of data types, and does not have collection data types, such as arrays, lists, or sets.

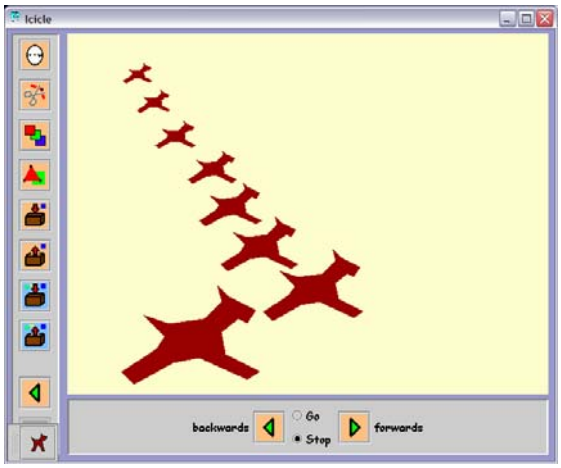


Figure 100 – The same performer type at different scales (and speeds).

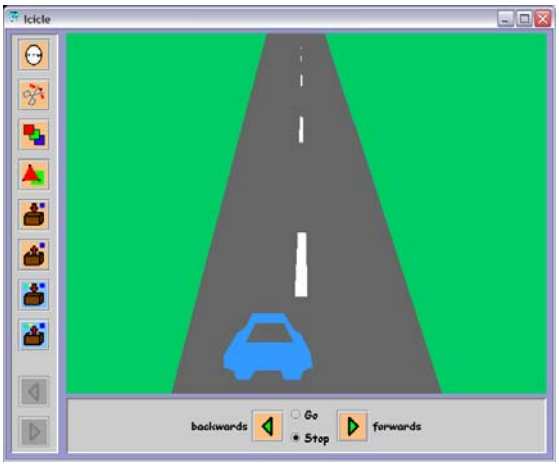


Figure 101 – The road appears to move towards the player.

Three-D programs

These days many game programs simulate a three-dimensional environment. There are even some educational programs or environments for children (Robertson & Good, 2003) that use modern 3D game engines to produce realistic 3D graphics.

Icicle performers are two-dimensional, and Icicle has no facility to control 3D-graphics processors, but since Icicle scales operations according to the size of the performer, it can provide a 2½D feel to its programs. Larger objects appear closer, and move correspondingly faster across the world, giving the effect of parallax (see Figure 100). The effect is even more convincing if an object increases in size as it moves down the screen, or decreases in size as it moves up the screen, as the centre lines do in the racing car game in Figure 101.

Text on the screen

Text is not supported directly by Icicle. When Icicle was being designed, this was considered the most important capability excluded from the original version. A subsequent version must include the ability to place text on the world and to be able to manipulate text as values in Icicle variables. As was pointed out in Chapter 7, the children who worked with Icicle did not mention the lack of text presentation and processing, until it was pointed out to them in an evaluation session.

Producing text

If a child wants to display text with the current version of Icicle, it can be produced using performers. This may sensibly be done in at least two ways. One performer can be created for every letter of the alphabet. This is tedious and every letter has to have its own set of rules. Another approach is to have one alphabet performer that can morph to any desired letter. One shape is added to the performer for each letter (or character) it is to represent. A variable can be used to control the performer's appearance. To change an "A" to a "Z" replace the 1 in the *letter* variable with a 26. The advantage of this technique is that the letters share the same rules and so the rules only have to be defined once.

These approaches are satisfactory if you want to put small messages into the world. It quickly becomes painful if you want to display a lot of text and each letter has to be dropped onto the world. A better technique is to make a cursor performer that receives all possible key presses. When a key is pressed, the cursor performer creates an alphabet performer at its current position, tells the alphabet performer to morph to the correct key shape, e.g., an "A", and then moves one character's width across the screen.

Figure 102 shows a simple text editor, which employs this technique. Only two performer boxes were needed for this program – the cursor and the alphabet performer boxes. The cursor moves backwards when the delete key is pressed, and any characters it collides with delete themselves. It is simple to make the cursor move down a line when it reaches the edge of the screen by modifying the standard rule for the cursor's *across* variable.

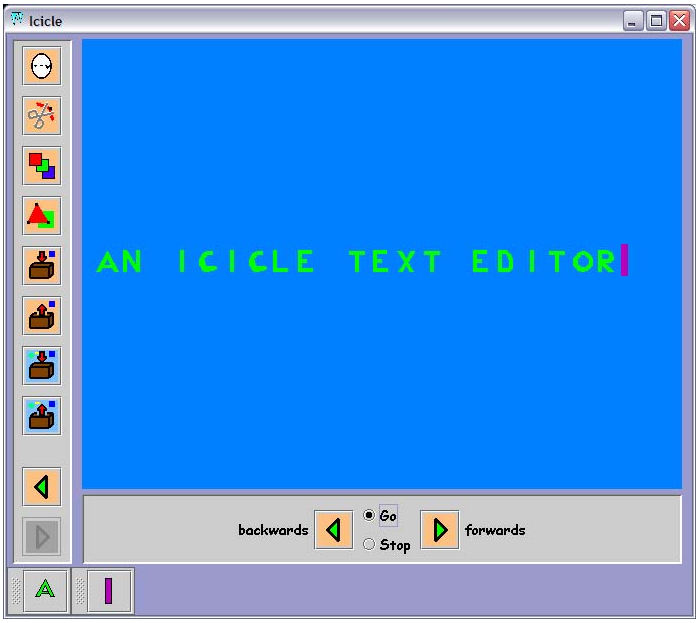


Figure 102 – A simple text editor written in Icicle.

Data types

The only data types that Icicle provides are performer boxes, performers, numbers, shapes, colours, sounds, and star messages. Every performer box is the template or class for multiple performers or instances. Performer boxes can only be manipulated at the user interface level; they cannot be controlled from within an Icicle program. Performers are the only true objects in Icicle. As we have seen, nothing happens in an Icicle program unless there is a performer or an associated variable to carry out the instruction.

Performers are used as parameters in some instructions, e.g., *create* and *tell*. They are also referred to in rule conditions, e.g., *when created by*, *when collides with*, *when no longer touches*.

Numbers are used in several different ways in Icicle. Many of the performer commands include a number as a parameter, e.g., moves and turns. In many of these cases, numbers are used, not only for the amount of the instruction – the number of steps to move, or the number of degrees to turn through – but also for the speed the instruction is to work at. In the current version of Icicle, these numbers can only be altered directly by the user, as they are not Icicle variables; from within an Icicle program they function as constants or literals. The numbers can be changed in the instruction editors (Chapter 5).

Numbers also appear as constants within variable assignment expressions, and in the *when* conditions of variable rules. Numbers are also stored in variables. The only variables provided by Icicle are floating-point values (disguised as integers when possible, to look less intimidating).

The remaining data types - shapes, colours, sounds and star message names are only functional as parameters to specific instructions. Shapes can be modified from within Icicle programs with the *morph* and the *zoom* instructions, or by setting the *zoom* variable. Colours are one aspect of the shape data type and can be changed with *morph* instructions. Sounds are employed by the *play* instruction, and star message values are used when sending messages.

Data collections

Given that Icicle does not have array, list or vector data types, it is still possible to provide some of the operations that work on such structures, using a number of performers. Here is a method that can be used in many situations:

- Use one performer to create the required number of performers.
- Repeat as necessary
    - Send a message to all created performers, asking them to perform a certain operation, e.g., report a variable value or make a comparison.
    - If there is a response, do something with it.

As an example, the following Icicle program generates 20 random numbers and finds the maximum and minimum values. Figure 103 shows the setup to the program.

- At the top-left of the figure, the *M*-shaped performer (called the *M*-performer) initialises three variables: the *create count* variable is set to 20 (the required number of performers), the *maximum* variable is set to 0 and the *minimum* variable is set to 101.
- Setting the *create count* variable to 20 starts a loop to create 20 green, kite-shaped performers (called kite performers). At the bottom of Figure 103, the rules for the *create count* variable show that when a value greater than zero is assigned to the variable, the *M*-performer creates a kite performer and moves 30 steps to the right. The move is unnecessary, but does mean that the created performers are not on top of each other. The last thing the rule does is to decrement the *create count* variable.
- The kite performers have a variable called *value*. When a kite performer is created (the rule at the top-right of Figure 103), this variable is set to a random number between 1 and 100 inclusive.
- After twenty kite performers have been created (in the *when = 0* rule of the *create count* variable, at the bottom of Figure 103), the *M*-performer sends two messages to all kite performers, starting the minimum and maximum calculations in parallel.

The calculations themselves are carried out with a combination of variable assignments and messages (Figure 104).

- When the kite performers receive the messages, they assign the maximum and minimum values from the *M*-performer into comparison variables, *max comparison* and *min comparison* (the top-left of Figure 104). We will only follow the maximum calculation here; the minimum calculation carries on in parallel and is essentially the same.
- If the incoming value is less than the number in the kite performer's *value* variable, the performer sends a message back to the *M*-performer, indicating that a greater value is available (the bottom-left of Figure 104). All kite performers will reply the first time sending the *orange 5* message to the *M*-performer.
- If the incoming value is greater than or equal to the *value* variable the kite performer does nothing. No message goes to the *M*-performer.

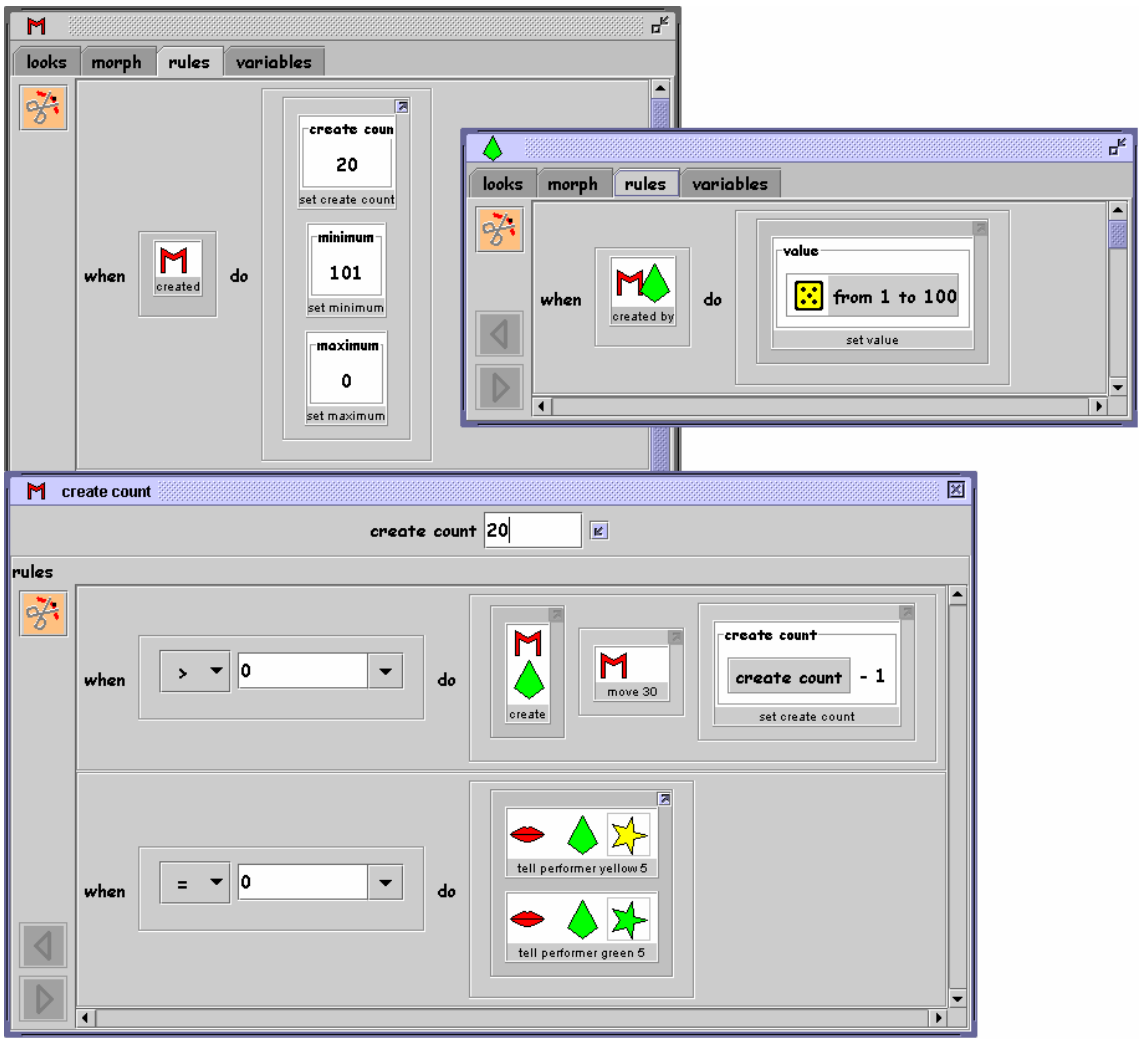


Figure 103 – The setup to the max and min program.

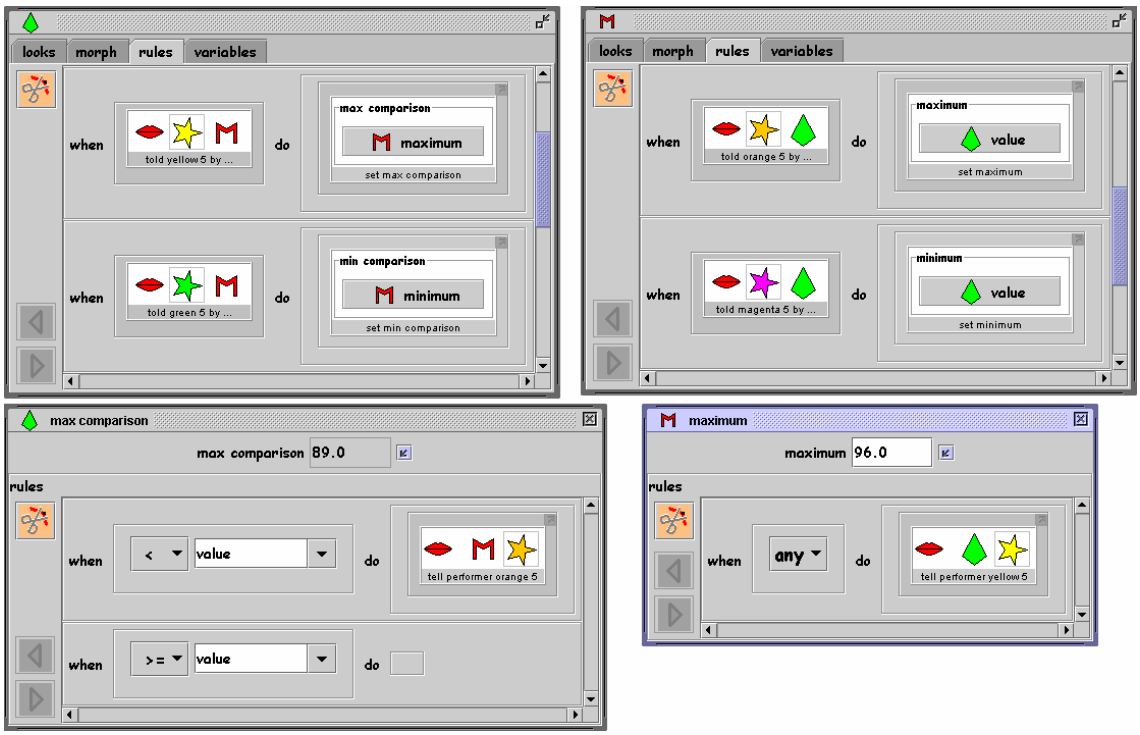


Figure 104 – Calculating the maximum value.

- The messages that are sent result in one kite performer's *value* being assigned to the *M*-performer's *maximum* variable (the top-right of Figure 104). Only the last assignment of all those in the same clock-tick has any effect.
- This assignment to the *maximum* variable (the bottom-right of Figure 104) starts the next iteration of message passing. Except, this time around, the *maximum* value will be larger than it was and fewer kite performers will reply.
- When no replies come back from the kite performers, the *maximum* variable must hold the maximum value.

In the worst case, there will be the same number of iterations as there are kite performers. In the best case, there will be only one iteration.

The number of messages sent in these maximum and minimum calculations is $O(n^2)$, where n is the number of kite performers. This seems painfully high as a simple scan through an array in most programming languages would be O(n). In reality, the number of performers is low and each kite performer is receiving and sending messages in parallel. There are only O(n) iterations of messages sent from the *M* performer and so only O(n) Icicle clock-ticks are used for the calculations.

The reason for the n^2 term is that the messages are broadcast from the *M* performer to all kite performers during each iteration. We can avoid this if we process the kite performers in turn rather than in parallel. One way of doing this is to get the value from each kite performer as it is created. Another technique is to make the *M* performer collide with each kite performer, one after another, collecting the maximum and minimum values as it goes. This collision technique can be usefully exploited by other algorithms, such as sorts.

Figure 105 shows a sort program executing using this technique. The colour of the performers matches the value; a deeper red represents a higher value. Performers move to the left or the right depending on the values of their neighbours when they collide with them. Figure 106 shows the completed sort. The number of collisions between performers in this program is $O(n^2)$ (not surprising when you watch the program execute, as lighter-coloured performers bubble to the left) but the number of Icicle clock-ticks to completion is O(n) as you would expect with a parallel bubble-sort (effectively an odd-even transposition sort (Bitton, DeWitt, Hsaio, & Menon, 1984)).

Figure 105 – A row of partially sorted performers.

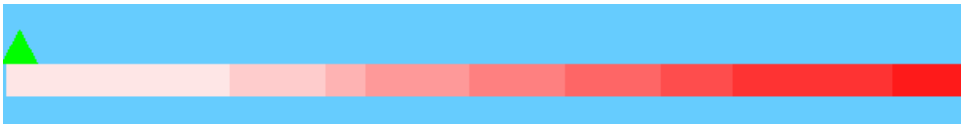


Figure 106 – The completely sorted performers.

Program-modifying programs

Some computer languages provide the ability to modify themselves whilst running, adding extra functions or altering initial behaviours. This is sometimes referred to as a meta-level of programming since it is programming the program itself. Many of the languages referred to as *scripting languages*, such as Python (Python Software Foundation) and Ruby (Thomas & Hunt, 2001), do this easily, as do Smalltalk (Goldberg, 1984) and Lisp (Winston & Horn, 1989).

In Icicle, there is no way to create or modify the rules associated with a performer via another rule. Rules can only be changed by the user at the interface level. This prevents meta-level programming. Icicle is not alone in this limitation; most compiled languages do not allow this level of control.

In the world of programming languages for children, Logo and its descendents have always provided meta-level programming with the *run* command that accepts any list as a possible sequence of instructions. As we saw, Logo was designed as a simplified version of Lisp. Apart from Squeak, which is built on Smalltalk, none of the other widely available programming languages for children provides this capability.

Rather than considering programs that modify themselves, we can also talk about programs that produce other programs. A standard test for many computer languages is being able to implement a compiler or interpreter for the language in the language itself. Theoretically, Icicle could be used to produce an interpreter for Icicle programs, but in reality, the number of rules required would make it infeasible.

COMPUTATIONAL POWER

If we ignore problems of speed and the awkwardness of implementation, are there computations that can be made with other programming languages that cannot be made with Icicle?

Turing machine equivalence

Loosely, the Church-Turing thesis states that anything that can be effectively calculated can be carried out on a Turing machine (D. I. A. Cohen, 1986). It turns out that implementing Turing machines in Icicle is straightforward.

Turing machine

A Turing machine consists of a tape head with a tape running through it and a number of instructions that are carried out according to a simple decision table.

1. The machine can be in one of a finite number of conditions or states.
2. The tape running through the tape head consists of square sections each capable of containing one symbol.
3. The tape head scans one symbol from the tape at a time.
4. Depending on the symbol scanned and the current state of the machine different instructions or actions are performed.
5. The tape head can erase the scanned square of tape or write a new symbol there.
6. The machine can change its state.

7. The machine can move the tape one section to the left or one section to the right through the tape head.

Correspondences

An Icicle Turing machine uses performers for both the tape head and for each section of the tape. Rather than moving the tape to the left and right through the tape head, it is easier if the tape head moves backwards and forwards over the tape. In Figure 107, the triangle performer is the tape head. The tape is represented by the black and white squares. In this case, different colours are used to display the different symbols on the squares of tape.

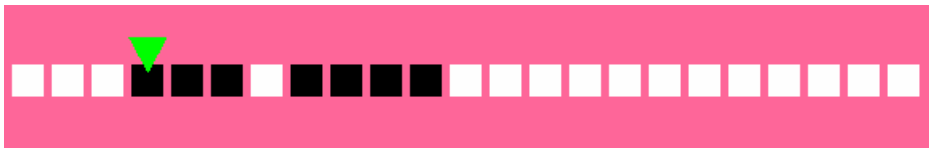


Figure 107 – A Turing machine implemented in Icicle.

1. The state of the machine is represented as the value of a variable associated with the tape head performer.
2. The tape is represented by a number of performers created in a line. By switching off the wrap-around rule as many squares as are necessary can be created. (This is not entirely true, see below.) The symbol on a square of tape is stored in a variable of the square performer, and colours (or shapes) can be used to display the current symbol to the user.
3. The tape head scans the performer it has just collided with. The squares are spaced so that the tape head will only collide with one square at a time.
4. The value scanned from the tape and the value of the state of the machine are used to jump to a particular action by assigning a number to a variable, called *result state* in Figure 108.

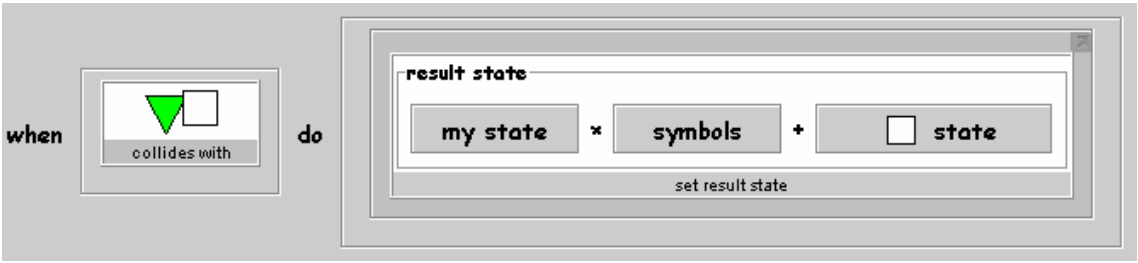


Figure 108 – Calculating the state after a collision.

The instructions in the *result state* rules correspond to the possible actions of the Turing machine.

5. A message can be sent to the scanned square of tape to tell it to change its symbol (and hence colour).
6. The machine state is changed simply by assigning a value to the state variable.
7. The head moves backwards and forwards along the tape with ordinary performer move instructions.

Figure 109 shows one such group of instructions. In this case, the head moves backwards on the tape, changes the machine's state to 8 and tells the current square to modify its symbol to 1.

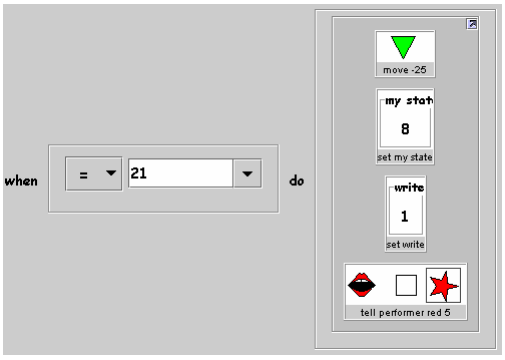


Figure 109 – A state change as a result of examining a section of tape.

A Turing adding machine

As an example, a simple Turing adding machine was implemented in Icicle using these correspondences. The machine used three symbols on the tape: black represented 1, white represented 0 and red was used as a marker to indicate where the calculation was up to. There were 11 states for the machine, and 25 rules in the decision table, depending on the state and the current symbol. Figure 110 shows the machine as it is adding 3 to 4. The answer is being written to the tape on the right hand side. The red square after the two black squares on the left is the marker symbol showing the current position of the calculation.

Figure 110 – Part way through calculating 3 + 4.

Figure 111 shows the result of the completed program. The head performer is not visible as the last instruction moves the head off the tape.

Figure 111 – 3 + 4 = 7.

A Universal Turing Machine

In order to prove that Icicle is as general as any other programming language it should be shown that it is equivalent to Turing's universal computing machine (Turing, 1936), now referred to as a Universal Turing Machine (UTM). This machine is the one that can simulate the behaviour of any other Turing machine by accepting an encoding of the instructions of the other Turing machine as a section of data on the tape.

We have seen how Icicle can implement a Turing machine, but Turing machines can have an infinite amount of tape whereas the implementation of Icicle has a limit to the number of performers that can be added to an Icicle world. This limit was chosen to provide acceptable performance on machines of moderate processing power. The limit could be removed to implement larger Turing machines. Of course, all other real programming systems have a limit on memory size as well. Similarly, any implementation of Icicle will only have memory space for rules associated with a specific number of states; in the current implementation, this is associated with the amount of memory allocated to the Java process when Icicle is run.

Since UTMs are specific Turing machines and we have seen a technique to represent any Turing machine in Icicle, it is possible to represent any UTM as well. The numbers of states and symbols do not even have to be very large. In Minsky's version of a UTM (discussed in (Feynman, Hey, & Allen, 1996)) there are only 8 symbols, and 23 states. In this specific, theoretical way, Icicle is as powerful as any other programming language and computing system.

IMPROVEMENTS TO ICICLE

There is no such thing as the perfect programming environment for children. Even the best ones have difficulties that make it hard for children to do the things they want to do with them. As has been shown, Icicle is better in some ways than other programming environments, but there are many ways in which it is still too difficult.

Control structures

- The conditions for rules are currently single interactions. This makes it complicated to make a rule execute when multiple conditions occur. A technique like that employed in AgentSheets could be used.
- A simpler interface should be provided for loops. Currently, loops are produced by the last instruction in a rule causing an interaction. Some method similar to that used in ICE could be employed for simple loops.
- There needs to be a simpler interface to control the situations where an interaction stops one rule immediately and starts another one. Currently, this is done with assignments to variables.

Variables

- Variables only represent numbers. They should be extended to represent extra types, especially text.
- All parameters to instructions are constant; they cannot be altered from an Icicle rule. All instructions should be able to use variables as parameters.

Messages

- Message interactions are merely event notifications. If a receiver requires information, it must inspect the variables of the sender. A way of specifying message content with the message would be cleaner.

Programming support

- There needs to be more debugging support. A way to represent the rules that are active and the instructions they are up to when single stepping, would have helped to find Edna's bug in Chapter 7.

User interface

- The Icicle user interface uses the standard Java controls and components. These are not the best controls for children and in one case did not provide the required functionality
- There are several minor interface problems, such as highlighting objects on drag-over, that should be corrected.

CONCLUSION

Any careful study of a new interface necessarily touches on a wide range of topics, from psychology to software design. In Icicle, the range is even wider, including educational psychology and process management. Within this range, Icicle embodies significant developments at different levels. There are contributions in the area of programming by demonstration environments, especially with regards to programming environments for children. There are also other contributions, reminding us that programming is something that children can do.

REDUCING THE NUMBER OF ABSTRACTIONS

Icicle is a programming language for children, designed to allow the production of simple games and simulations. It was aimed at children between the ages of seven and eleven. This age range coincides with Piaget's concrete operations stage, where children need information to be represented concretely, thus, there was an attempt to remove unnecessary levels of abstraction in the environment and in the way children interact with the environment as they construct programs.

As a simulation environment, Icicle represents objects that can move around the screen, interacting with the users and each other. A simulation of objects moving around the screen is more realistic, and hence less abstract, if the objects can move smoothly from one position to any other. Similarly, a screen object is perceived as more realistic if it can make smooth changes in appearance, orientation, and size. Richer and more complex worlds can be represented by a system that allows such changes.

Other abstract concepts, such as instructions and parallelism, are represented in Icicle in a concrete manner. The animation of instruction icons is used to show the actions to be performed by the instructions, and parallel instructions are produced by dropping instructions on top of each other.

These mechanisms lead to the four major contributions the Icicle system makes concerning the production of programs and how they are represented in programming by demonstration environments: the removal of the restrictions that grid-based systems have on the placement, orientation and size of programmable objects, the representation of rules with animations, the way in which parallelism is produced and the level at which it works, and the method of producing the program rules when new states arise.

A movement from discrete to continuous systems

A major factor in the removal of abstraction was the move from discrete representations of program objects in space and time, to continuous representations. The systems closest to Icicle, the Stagecast Creator (D. C. Smith et al., 2000) and AgentSheets (Repenning, 1993) PBD environments, divide the programmable area into rectangular grids and require objects to occupy one or more grid spaces. Icicle performers can be placed anywhere within an Icicle world (page 64). This freedom is consistent with perceiving the performers as being more realistic.

Similarly, Icicle performers can be turned to face in any direction. In many situations this means that only one shape or appearance needs to be constructed in order to show the performer facing different directions.

With grid-based systems an object can only move from one grid rectangle to another. As Icicle has no restriction on placement or orientation there is nothing to prevent a performer moving from any position to any other. The smoothness of any move depends on the speed at which the performer is moving.

Icicle performers can be stretched or shrunk to any size. The movement and creation instructions scale correctly along with the size of the performers.

Many Icicle instructions happen over a period of time, rather than in a single tick of the program clock (page 121). This is another move to seeing the program as a series of continuous actions, and hence smoother and more realistic. Extending instructions over a duration is also necessary for the Icicle approach to parallelism.

These additions allow more complex worlds, than allowed by existing systems. The downside of this is the increased processing requirements, both for interactions such as collisions and the scheduling system.

Representing instructions with animations

All Icicle instructions are represented by animated icons (page 71). Most of these, including the *move*, *turn*, *morph*, and *create* animations, explicitly show the effects of the instruction. These explicit representations made it easier for children to describe correctly the actions of instructions by *reading* the rule sequences. In particular *turn* instructions and instructions dependent on orientation were described accurately. This new visualisation of rules was essential, given the increased complexity of actions with continuous valued arguments.

The animations for instructions and the conditions of rules are activated one after the other, conveying the sequential nature of instruction execution, leading the user's eyes and removing unnecessary visual distractions. This was important in helping children understand complex rules.

Producing parallelism

To make Icicle simulations more realistic they have to allow parallel activities. In general, parallelism is considered an advanced programming feature, but the visualisation and demonstration techniques employed in Icicle put parallelism within reach of children. It provides simple techniques to enable parallelism within and between rules. These techniques work at the level of individual instructions and the level of rules themselves.

Instruction level

Icicle provides a novel technique for enabling users to demonstrate parallel instructions. Dropping the iconic representation of one instruction on top of another means the two instructions are to operate in parallel (page 105). Icicle provides two alternative visualisations of this. The resulting parallel instruction icon is animated, showing exactly what the new combined instruction does. The

instruction can also be expanded to show its constituent instructions animated in parallel. These parallel instructions are most effective with the instructions that execute over a duration of several clock ticks. Children employed this level of parallelism frequently, sometimes in novel and unexpected ways (page 137).

Instruction-stream level

Similar PBD environments do not execute instruction streams in parallel. The instructions from one stream complete before the instructions from another begin. Icicle allows multiple instruction streams to act on a performer simultaneously. This was used to good effect by children, when pushing two keys simultaneously to drive a performer in smooth curves around the screen, for example.

Allowing parallelism, of both kinds, also extends the range of possible simulations. The use of parallelism combines particularly well with the extension to a continuous world where actions have durations, and allows very rich kinds of movements.

Producing the program rules

A major difficulty with programming by demonstration is identifying when the user has to demonstrate additional rules and when the program should simply run. Requiring the user to control this decision completely puts too great a conceptual load on the user, especially if the user is a child. Icicle demonstrates an approach that reduces this burden.

The system actively asks the user to make a new rule when a change in state occurs that does not match a condition in the existing list of rules. PBD techniques can then be used to produce the rule instructions. Actions converted into instructions can be undone and redone, changing both the list of stored instructions and the state of the running program.

This production of rules when prompted was used successfully by children. Feedback from children showed that this technique of prompting whenever there is no existing condition can result in too many prompts (page 89). Heuristic decisions that restrict the prompting to just certain categories of conditions, and automatically provide default rules for other categories, provide an effective balance. All rules can be edited any time after they are defined.

OTHER CONTRIBUTIONS

Icicle, and ICE before it, made programming easy for children. The removal of abstraction mentioned above was one aspect of this. Another aspect was that the environments were designed to convey extra information about programming concepts to the users.

ICE revealed loops by automatically recognizing repeated actions, and procedures by collecting sequences of instructions into single commands with one mouse drag. Icicle used sequences of animations to convey the sequential nature of instructions. The layout of instructions in Icicle rules clearly revealed the differences between sequential and parallel instructions.

The programming model used in Icicle was conveyed by the technique to create rules. The user's attention was captured when a new rule had to be created. This prompted the user to make decisions

about the behaviour of program objects. It also revealed the underlying control mechanism of Icicle – rules are activated when states change.

To evaluate the conceptual understandings promoted by these design decisions was beyond the scope of this work, but children successfully used the environments to produce their own programs. This means that two further claims can reasonably be made.

Useful guidelines were produced to help design a programming environment for children

Twelve guidelines were used to direct the production of Icicle. These guidelines were shown to be successful to the extent that Icicle matched the abilities of children, and enabled them to produce programs from their own designs. Future programming environments for children could benefit from an inspection of these guidelines.

Programming is something that children can do

In the years since the early 1990s, and the demise of Logo in schools, most children have not been exposed to computer programming. This work is a reminder that programming is something that children can do, and that many children *want* to program, or at least enjoy it when provided with an appropriate environment. In the study with nine and ten-year-old children, ten out of twelve of them worked with Icicle enthusiastically. With modern environments, such as Stagecast Creator, ToonTalk and Icicle, it is possible to give a new generation of children the power to create their own programs.

BIBLIOGRAPHY

Abelson, H., & DiSessa, A. A. (1981). *Turtle geometry : the computer as a medium for exploring mathematics*. Cambridge, MA: MIT Press.

Agalianos, A. S. (1996, April 8-12). *Towards a Sociology of Educational Computing*. Paper presented at the Annual Meeting of the American Educational Research Association, New York, NY.

Anderson, C. A., & Dill, K. E. (2000). Video Games and Aggressive Thoughts, Feelings, and Behavior in the Laboratory and in Life. *Journal of Personality and Social Psychology, 78*(4), 772-790.

Armstrong, A., & Casement, C. (2000). *The Child and the Machine: How Computers Put Our Children's Education at Risk*: Robins Lane Press.

Assumpcao, J. (2000). *Pegasus 2000 - A Computer for Children*, from http://www.merlintec.com/pegasus2000/

Bacon, J., & Harris, T. (2003). *Operating systems: concurrent and distributed software design*. Harlow: Pearson Education Limited.

Baecker, R., Small, I., & Mander, R. (1991, 27 April-2 May). *Bringing icons to life*. Paper presented at the Human Factors in Computing Systems, Reaching Through Technology, CHI '91, New Orleans, LA, USA.

Baer, D., Groenewoud, P., Kapetanios, E., & Keuser, S. (2001, 31 May-1 June). *A semantics based interactive query formulation technique*. Paper presented at the Second International Workshop on User Interfaces in Data Intensive Systems, UIDIS 2001, Los Alamitos, CA, USA.

Barba, R. H. (1990). Assessing Children's Attitudes Towards Computers: The Draw-A-Computer User Test. *Journal of Computing in Childhood Education, 2*(1), 5-17.

Beaudouin-Lafon, M., Mackay, W. E., Andersen, P., Janecek, P., Jensen, M., Lassen, M., et al. (2001, 25-29 June). *CPN/Tools: a post-WIMP interface for editing and simulating coloured Petri nets*. Paper presented at the Applications and Theory of Petri Nets 2001, 22nd International Conference, Newcastle upon Tyne, UK.

Bitton, D., DeWitt, D. J., Hsaio, D. K., & Menon, J. (1984). A taxonomy of parallel sorting. *ACM Computing Surveys, 16*(3), 287-318.

Blackwell, A. F. (2001). Pictorial representation and metaphor in visual language design. *Journal of Visual Languages & Computing, 12*(3), 223-252.

Blackwell, A. F. (2002, 3-6 Sept.). *First steps in programming: a rationale for attention investment models*. Paper presented at the IEEE Symposia on Human Centric Computing Languages and Environments, Arlington, VA, USA.

Blackwell, A. F., Britton, C., Cox, A., Green, T. R. G., Gurr, C., Kadoda, G., et al. (2001, 6-9 Aug). *Cognitive dimensions of notations: design tools for cognitive technology*. Paper presented at the Cognitive Technology: Instruments of Mind. 4th International Conference, CT 2001, Coventry, UK.

Blackwell, A. F., & Burnett, M. (2002, 3-6 Sept.). *Applying attention investment to end-user programming*. Paper presented at the IEEE Symposia on Human Centric Computing Languages and Environments, Arlington, VA, USA.

Block, E. B., Simpson, D. L., & Reid, D. (1987). Teaching young children programming and word processing skills: The effects of three preparatory conditions. *Journal of Educational Computing Research, 3*(4), 435-442.

Brandes, A. A. (1996). Elementary School Children's Images of Science. In Y. Kafai & M. Resnick (Eds.), *Constructionism in Practice: Designing, Thinking and Learning in a Digital World*. Mahwah, New Jersey: Lawrence Erlbaum.

Broderbund. (1991-2003). KidPix: Riverdeep.

Brosnan, M. J. (1999). A new methodology, an old story? Gender differences in the "draw-a-computer-user" test. *European Journal of Psychology of Education, XIV*(3), 375-385.

Bruner, J. S. (1966). *Toward a theory of instruction*. Cambridge, MA: Belknap Press of Harvard University.

Burnett, M., Atwood, J., Djang, R. W., & Reichwein, J. (2001). Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming, 11*(2), 155-206.

Cardelli, L., & Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv., 17*(4), 471-523.

Carvalho, J. (2000). Using AgentSheets to teach simulation to undergraduate students. *Journal of Artificial Societies and Social Simulation, 3*(3).

Chang, S. K. S. K., Ichikawa, T., & Ligomenides, P. A. (Eds.). (1986). *Visual languages*. New York: Plenum Press.

Clements, D. H. (1983/84). Supporting young children's Logo programming. *The Computing Teacher, 11*(5), 24-29.

Clements, D. H. (1995a). Playing with computers, playing with ideas. *Educational Psychology Review, 7*(2), 203-207.

Clements, D. H. (1995b). Teaching creativity with computers. *Educational Psychology Review, 7*(2), 141-161.

Clickteam. (1994). Klik & Play: Clickteam.

Cockburn, A., & Bryant, A. (1997). Leogo: an equal opportunity user interface for programming. *Journal of Visual Languages & Computing, 8*(5-6), 601-619.

Cockburn, A., & Bryant, A. (1998, 15-17 July). *Cleogo: collaborative and multi-metaphor programming for kids*. Paper presented at the 3rd Asia Pacific Computer Human Interaction, Los Alamitos, CA, USA.

Cohen, D. I. A. (1986). *Introduction to Computer Theory*: John Wiley & Sons, Inc.

Cohen, J. D., Lin, M. C., Manocha, D., & Ponamgi, M. (1995, 9-12 April). *I-COLLIDE: an interactive and exact collision detection system for large-scale environments*. Paper presented at the Symposium on Interactive 3D Graphics, Monterey, CA, USA.

Cohen, R. (1987). Implementing Logo in the Grade Two Classroom: Acquisition of Basic Programming Concepts. *Journal of Computer-Based Instruction, 14*(4), 124-132.

Cohen, R., & Geva, E. (1989). Designing Logo-like Environments for Young Children: The Interaction between Theory and Practice. *Journal of Educational Computing Research, 5*(3), 349-377.

Cordes, C., & Miller, E. (2000). *Fool's Gold: A Critical Look at Computers in Childhood*: Alliance for Childhood.

Crook, C. (1992). Young children's skill in using a mouse to control a graphical computer interface. *Computers & Education, 19*(3), 199-207.

Cypher, A. (Ed.). (1993). *Watch what I do : programming by demonstration*. Cambridge, MA: MIT Press.

Dahl, O.-J., Myhrhaug, B., & Nygaard, K. (1970). *The SIMULA 67 common base language* (Technical report No. S-22). Oslo, Norway: Norwegian Computing Center.

Davis, R., & King, J. (1984). The Origin of Rule-Based Systems in AI. In B. G. Buchanan & E. H. Shortliffe (Eds.), *Rule Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project* (pp. 20-52). Reading, MA: Addison-Wesley Publishing Company, Inc.

Denham, P. (1993). Nine to fourteen-year-old children's conception of computers using drawings. *Behaviour and Information Technology, 12*(6), 346-358.

DiSessa, A. A., & Abelson, H. (1986). Boxer: a reconstructible computational medium. *Communications of the ACM, 29*(9), 859-868.

Druin, A. (1999). *Cooperative inquiry: developing new technologies for children with children*. Paper presented at the SIGCHI conference on Human factors in computing systems, Pittsburgh, Pennsylvania, United States.

du Boulay, B. (1989). Some difficulties of learning to program. In E. Soloway & J. C. Spohrer (Eds.), *Studying the Novice Programmer*. Hillsday, NJ: Lawrence Erlbaum.

Duff, M. J. B. (1980). Array processing. *IEE Review, 26*(11), 888-889, 891-883.

Egan, K. (2002). *Getting it Wrong from the Beginning: Our Progressivist Inheritance from Herbert Spencer, John Dewey, and Jean Piaget*. New Haven: Yale University Press.

Fay, A. L., & Mayer, R. E. (1987). Children's Naive Conceptions and Confusions About Logo Graphics Commands. *Journal of Educational Psychology, 79*(3), 254-268.

Feynman, R. P., Hey, A. J. G., & Allen, R. W. (1996). *Feynman lectures on computation*. Reading, MA: Addison-Wesley.

Finzer, W., & Gould, L. (1984, June). Programming by Rehearsal. *BYTE, 9,* 187-210.

Gage, N. L., & Berliner, D. C. (1984). *Educational psychology* (3 ed.). Boston: Houghton Mifflin.

Gallimore, R., & Tharp, R. (1990). Teaching mind in society: Teaching, schooling, and literate discourse. In L. C. Moll (Ed.), *Vygotsky and Education: Instructional Implications and Applications of Sociohistorical Psychology*: Cambridge University Press.

Gilmore, D. J., Pheasey, K., Underwood, J., & Underwood, G. (1995, 27-29 June). *Learning graphical programming: An evaluation of KidSim*. Paper presented at the Human-Computer Interaction, Interact '95, Lillehammer, Norway.

Goldberg, A. (1984). *Smalltalk-80 : the interactive programming environment*. Reading, MA: Addison-Wesley.

Goldstein, J. H. (Ed.). (1994). *Toys, Play and Child Development*. Cambridge: Cambridge University Press.

Gosling, J., Yellin, F., & The Java Team. (1996). *The Java Application Programming Interface* (Vol. 2). Reading, Massachusetts: Addison-Wesley.

Gostelow, K. P., & Thomas, R. E. (1980). Performance of a simulated dataflow computer. *IEEE Transactions on Computers, c-29*(10), 905-919.

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing, 7*(2), 131-174.

Greening, T. (1998). *Computer science: through the eyes of potential students.* Paper presented at the Third Australasian conference on Computer Science education, The University of Queensland, Australia.

Hanna, L., Risden, K., Czerwinski, M., & Alexander, K. J. (1999). The Role of Usability Research in Designing Children's Computer Products. In A. Druin (Ed.), *The Design of Children's Technology*. San Fransisco: Morgan Kaufmann.

Harada, Y., & Potter, R. (2003). *Fuzzy Rewriting - Soft Program Semantics for Children.* Paper presented at the IEEE Symposium on Human Centric Computing Languages and Environments, Auckland, New Zealand.

Harel, I. (1991). *Children designers : interdisciplinary constructions for learning and knowing mathematics in a computer-rich school*. Norwood, NJ: Ablex Pub. Corp.

Healy, J. (1999). *Failure to Connect: How Computers Affect Our Children's Minds - for Better and Worse*: Touchstone Books.

Howland, J., Laffey, J., & Espinosa, L. M. (1997). A computing experience to motivate children to complex performances. *Journal of Computing in Childhood Education, 8*(4), 291-311.

Hunt, A., & Kirk, R. (1999). *Digital Sound Processing for Music and Multimedia (Music Technology)*: Focal Press.

Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., & Kay, A. (1997). Back to the future. The story of Squeak, a practical Smalltalk written in itself. *ACM. Sigplan Notices (Acm Special Interest Group on Programming Languages). 32*(10), 318-326.

Kafai, Y. (1994). Electronic play worlds: Children's construction of video games. In Y. B. Kafai & M. Resnick (Eds.), *Constructionism in practice: Rethinking the roles of technology in learning*. Mahwah, NJ: Lawrence Erlbaum Associates.

Kafai, Y. (1996). Software by kids for kids. *Communications of the ACM, 39*(4), 38-39.

Kahn, K. (1996). ToonTalk-an animated programming environment for children. *Journal of Visual Languages & Computing, 7*(2), 197-217.

Kahn, K. (2004). Personal communication. Washington D.C.

Kurlander, D., & Feiner, S. (1992). *A history-based macro by example system.* Paper presented at the UIST. Fifth Annual Symposium on User Interface Software and Technology, New York, NY, USA.

LCSI. (2003). MicroWorlds EX.

Levin, B., & Barry, S. M. (1997). Children's Views of Technology: The Role of Age, Gender, and School Setting. *Journal of Computing in Childhood Education, 8*(4), 267-290.

Lewis, T., Rosson, M. B., Carroll, J., & Seals, C. (2002). *A community learns design: towards a pattern language for novice visual programmers.* Paper presented at the IEEE Symposia on Human Centric Computing Languages and Environments, Arlington, VA.

Lieberman, H. (Ed.). (2001). *Your Wish Is My Command: Programming by Example*: Morgan Kaufmann Publishers.

Lyman, E. R. (1978). *PLATO Highlights, Fifth Revision* (Descriptive): Illinois Univ., Urbana. Computer-Based Education Research Lab.[BBB01494].

Mayer, R. E., Dyck, J. L., & Vilberg, W. (1986). Learning to program and learning to think: what's the connection? *Communications of the ACM, 29*(7), 605-610.

McDaniel, R. G., & Myers, B. A. (1997, 14-17 Oct.). *Gamut: demonstrating whole applications.* Paper presented at the ACM Symposium on User Interface Software and Technology. 10th Annual Symposium. UIST '97, New York, NY, USA.

Möller, T., & Haines, E. (2002). *Real-time rendering* (E. Haines, Trans. 2nd ed.). Natick, MA: AK Peters.

Muller, M. J., & Kuhn, S. (1993). Participatory design. *Communications of the ACM, 36*(6), 24-28.

Myers, B. A., & McDaniel, R. G. (2001). Demonstrational Interfaces: Sometimes You Need a Little Intelligence, Sometimes You Need a Lot. In H. Lieberman (Ed.), *Your Wish Is My Command: Programming by Example*. San Fransisco: Morgan Kaufmann.

Norman, D. A. (1983). Some observations on mental models. In E. Gentner & A. L. Stevens (Eds.), *Mental Models* (pp. 7-14). Hillsdale NJ: Lawrence Erlbaum Associates.

Norman, D. A. (1986). Cognitve Engineering. In D. A. Norman & S. W. Draper (Eds.), *User Centered System Design* (Vol. 31-61). Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Pane, J. F., Myers, B. A., & Miller, L. B. (2002, 3-6 September). *Using HCI techniques to design a more usable programming system.* Paper presented at the IEEE Symposia on Human Centric Computing Languages and Environments, Arlington, VA.

Pane, J. F., Ratanamahatana, C., & Myers, B. A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies, 54*(2), 237-264.

Papert, S. (1980). *Mindstorms : children, computers and powerful ideas.* Brighton: Harvester.

Papert, S. (1993). *The children's machine: rethinking school in the age of the computer.* New York: BasicBooks.

Pea, R. D., & Kurland, D. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology, 2*(2), 137-168.

Perlin, K., & Fox, D. (1993, 1-6 Aug.). *Pad: an alternative approach to the computer interface.* Paper presented at the SIGGRAPH 20th Annual International Conference on Computer Graphics and Interactive Techniques. The Eye of Technology, Anaheim, CA, USA.

Piaget, J. (1930). *The Child's Conception of Physical Causality.* London: Routledge and Keegan Paul Ltd.

Python Software Foundation.*Python Programming Language*, from http://www.python.org/

Rader, C., Brand, C., & Lewis, C. (1997). *Degrees of Comprehension: Children's Understanding of a Visual Programming Environment.* Paper presented at the Human Factors in Computing Systems Conference, CHI 97, Atlanta, GA, USA.

Read, J., Gregory, P., MacFarlane, S., McManus, B., Gray, P., & Patel, R. (2002). *An investigation of participatory design with children - informant, balanced and facilitated design.* Paper presented at the International Workshop 'Interaction Design and Children', Maastricht, Netherlands.

Repenning, A. (1993). *Agentsheets: A tool for building domain-oriented visual programming environments.* Paper presented at the Conference on Human Factors in Computing Systems: INTERCHI'93, Amsterdam.

Repenning, A. (1995, 5-9 Sep.). *Bending the rules: steps toward semantically enriched graphical rewrite rules.* Paper presented at the 11th IEEE International Symposium on Visual Languages, Darmstadt, Germany.

Repenning, A., & Perrone, C. (2001). Programming by Analagous Examples. In H. Lieberman (Ed.), *Your Wish Is My Command: Programming by Example* (pp. 351-369): Morgan Kaufmann Publishers.

Repenning, A., & Sumner, T. (1995). Agentsheets: a medium for creating domain-oriented visual languages. *Computer, 28*(3), 17-25.

Resnick, M. (1994). *Turtles, termites, and traffic jams : explorations in massively parallel microworlds.* Cambridge, Mass.: MIT Press.

Reuters. (2004, 13 February). Microsoft Windows source code leaks onto internet. *The New Zealand Herald.*

Rieber, L. P. (1996). Animation as a Distractor to Learning. *International Journal of Instructional Media, 23*(1), 53-57.

Rieber, L. P., Smith, L., & Noah, D. (1998). The Value of Serious Play. *Educational Technology, 38*(6), 29-37.

Robertson, J., & Good, J. (2003). *Ghostwriter: a narrative virtual environment for children.* Paper presented at the 2003 conference on Interaction Design and Children, Preston, England.

Roschelle, J. M., Pea, R. D., Hoadley, C. M., Gordin, D. N., & Means, B. M. (2000). Changing how and what children learn in school with computer-based technologies. *Future of Children, 10*(2), 76-101.

Sandler, S. (1987, 22-24 Sept.). *Mixed-level simulation for system design and verification.* Paper presented at the Northcon/87 Conference, Los Angeles, CA, USA.

Scaife, M., & Taylor, J. (1990). Graduated learning environments for developing computational concepts in 7-11 year old children. *Journal of Artificial Intelligence in Education, 2*(2), 31-41.

Seals, C., Rosson, M. B., Carroll, J. M., Lewis, T., & Colson, L. (2002). *Fun learning Stagecast Creator: an exercise in minimalism and collaboration.* Paper presented at the IEEE Symposia on Human Centric Computing Languages and Environments, Arlington, VA.

Setzer, V. W., & Monke, L. (2001). Challenging the Applications: An Alternative View on Why, When and How Computers Should Be Used in Education. In R. Muffoletto (Ed.), *Education and Technology: Critical and Reflective Practices* (pp. 141-172). Cresskill, New Jersey: Hampton Press.

Sheehan, R. (1997, 14-19 June). *Designing a Web Browser for Children.* Paper presented at the World Conference on Educational Multimedia/Hypermedia: Ed-Media 97, Calgary, Canada.
Sheehan, R. (1999, 30 Aug.-3 Sept.). *Incremental control of a children's computing environment.* Paper presented at the International Conference on Human-Computer Interaction: INTERACT'99, Edinburgh, UK.
Sheehan, R. (2000, 10-13 Sept.). *Lower floor, lower ceiling: easily programming turtle-graphics.* Paper presented at the IEEE International Symposium on Visual Languages, Seattle, WA, USA.
Sheehan, R. (2002, June 24-29). *Turning ICE into Icicle.* Paper presented at the World Conference on Educational Multimedia, Hypermedia & Telecommunications: Ed-Media 2002, Denver, Colorado, USA.
Sheehan, R. (2003a). *Children's perception of computer programming as an aid to designing programming environments.* Paper presented at the Interaction Design and Children Conference: IDC 2003, Preston, England.
Sheehan, R. (2003b, 28-31 Oct.). *Parallelism in the Icicle programming environment.* Paper presented at the IEEE Symposium on Human Centric Computing Languages and Environments, Auckland, New Zealand.
Shneiderman, B. (1983). Direct manipulation: a step beyond programming languages. *Computer, 16*(8), 57-69.
Silberschatz, A., & Galvin, P. B. (1998). *Operating system concepts* (5th ed.). Reading, Mass.: Addison Wesley Longman.
Sloman, A. (1971). *Interactions between philosophy and artificial intelligence: The role of intuition and non-logical reasoning in intelligence.* Paper presented at the 2nd International Joint Conference on Artificial Intelligence, London.
Smith, D. C., Cypher, A., & Spohrer, J. (1994). KIDSIM: programming agents without a programming language. *Communications of the ACM, 37*(7), 54-67.
Smith, D. C., Cypher, A., & Tesler, L. (2000). Novice programming comes of age. *Communications of the ACM, 43*(3), 75-81.
Smith, G. G., & Grant, B. (2000). From Players to Programmers: A Computer Game Design Class for Middle-school Children. *Journal of Educational Technology Systems, 28*(3), 263-275.
Solomon, C. (1986). *Computer Environments for Children*. Cambridge, MA: The MIT Press.
Stoll, C. (1995). *Silicon snake oil : second thoughts on the information highway*. New York: Doubleday.
Tanimoto, S. (1990). VIVA: A visual language for image processing. *Journal of Visual Languages & Computing, 2*(2), 127-139.
The People's Embassy. (1998). *PROGENI SYSTEMS (NZ) LTD*, from http://www.embassy.org.nz/computer/progeni.htm
Thomas, D., & Hunt, A. (2001). *Programming Ruby - The Pragmatic Programmer's Guide*: Addison Wesley Longman.
Travers, M. D. (1996). *Programming with Agents: New metaphors for thinking about computation.* Unpublished PhD, Massachusetts Instititute of Technology, Cambridge, MA.
Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, 42*, 230-265.
Turkle, S. (1995). *Life on the Screen: Identity in the Age of the Internet*: Simon & Schuster.
Turkle, S., & Papert, S. (1991). Epistemological Pluralism and the Revaluation of the Concrete. In I. Harel & S. Papert (Eds.), *Constructionism*. Norwood, NJ: Ablex.
Ungar, D., & Smith, R. B. (1987). *Self: The Power of Simplicity.* Paper presented at the OOPSLA '87 Conference, Orlando, Florida.
Van Duuren, M., Dossett, B., & Robinson, D. (1998). Gauging Children's Understanding of Artificially Intelligent Objects: A Presentation of "Counterfactuals". *International Journal of Behavioural Development, 22*(4), 871-889.
various. (2001). *Wikipedia: The Free Encyclopedia*, from http://en.wikipedia.org
Vygotsky, L. S. (1978). *Mind in Society: The Development of Higher Psychological Processes*. Cambridge, MA: Harvard University.
Wiencek, H. (1987). *The World of LEGO Toys*. New York: Harry N. Abrams, Inc.
Wilensky, U. (1999). NetLogo: Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.
Winston, P. H., & Horn, B. (1989). *Lisp* (3rd ed.). Reading, MA: Addison-Wesley.
Wolber, D. W. (1997). Pavlov: an interface builder for designing animated interfaces. *ACM Transactions on Computer-Human Interaction, 4*(4), 347-386.

Wright, T., & Cockburn, A. (2000, 4-6 Dec.). *Writing, reading, watching: a task-based analysis and review of learners' programming environments.* Paper presented at the International Workshop on Advanced Learning Technologies: IWALT 2000, Palmerston North, New Zealand.

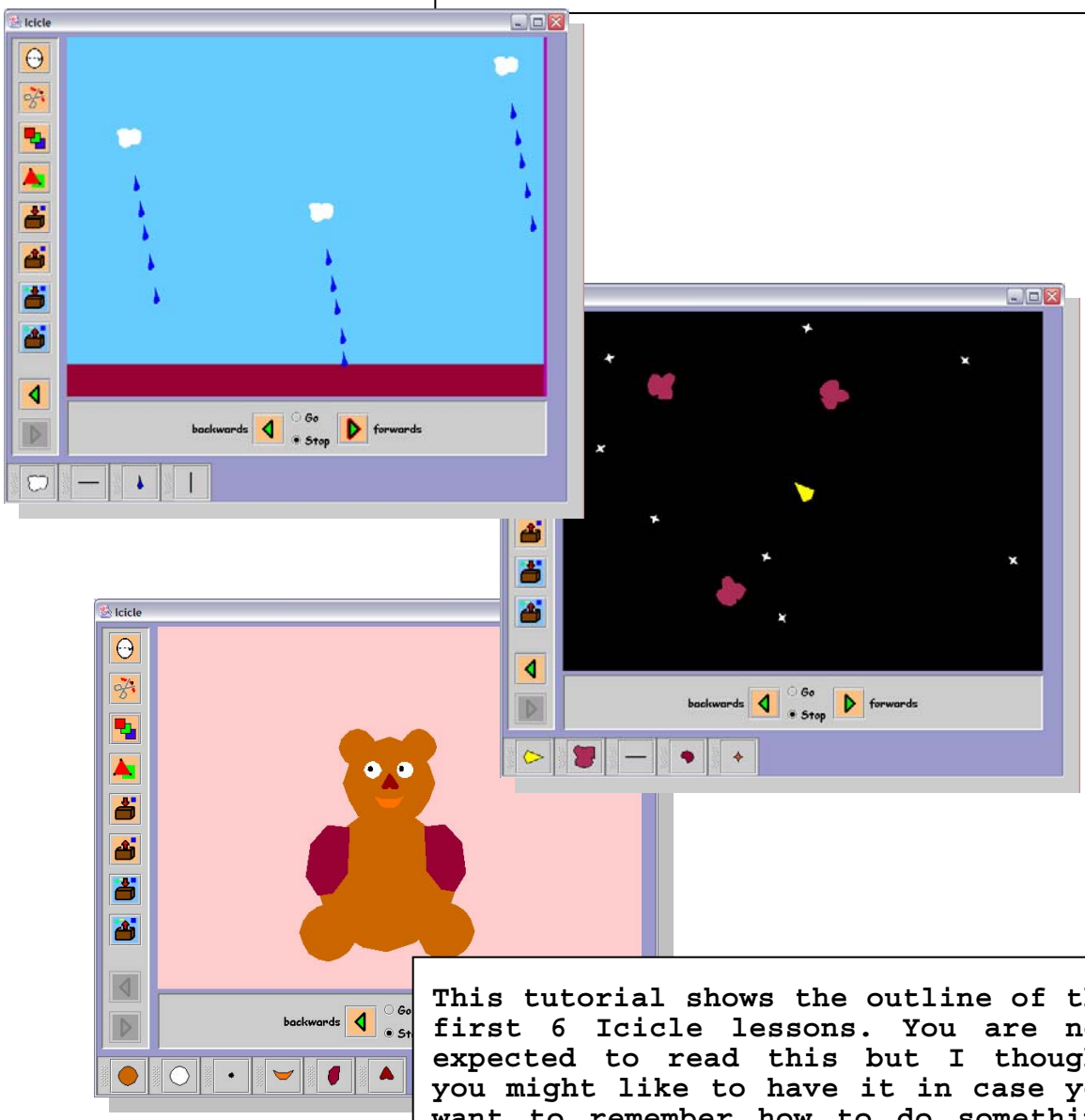
Yeshno, T., & Ben-Ari, M. (2001, April). *Salvation for Bricoleurs.* Paper presented at the 13th Workshop of the Psychology of Programming Interest Group: PPIG 13, Bournemouth UK.

APPENDIX A

This is the document provided to the children who used Icicle. As mentioned in Chapter 7, there are some minor differences between the version of Icicle represented in this tutorial and the version described in the thesis.

Your Icicle Tutorial

Write your name here:



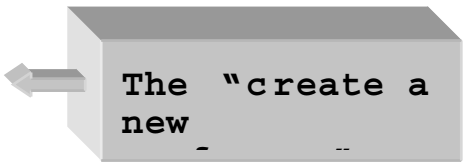
This tutorial shows the outline of the first 6 Icicle lessons. You are not expected to read this but I thought you might like to have it in case you want to remember how to do something in Icicle. I also thought that your parents might be interested.

Unfortunately, reading instructions

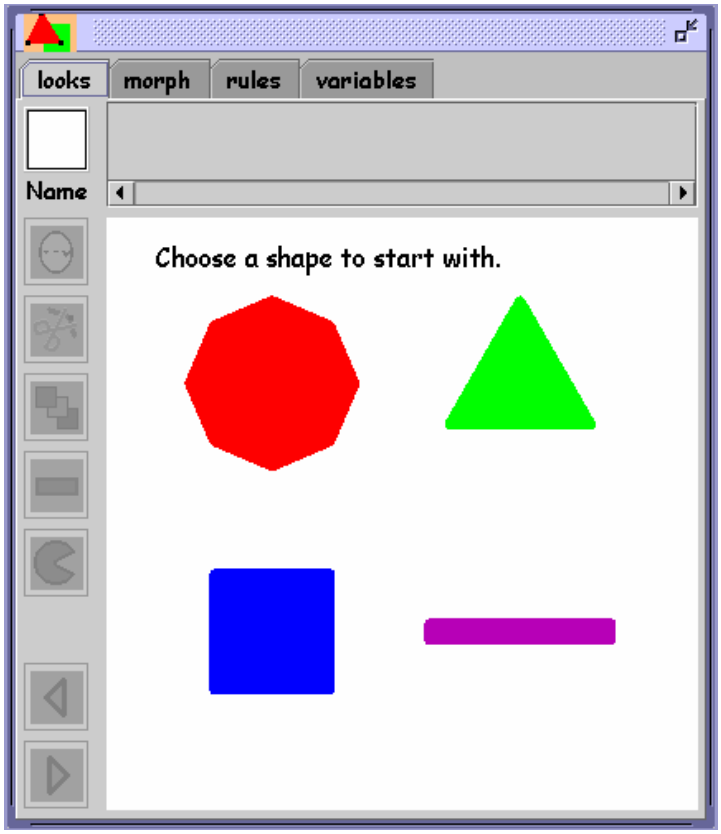
1.Introduction to the shape maker.

In an Icicle program everything is done by performers. A performer is a shape that can move around the screen and do other things. Before you do anything else you have to make some performers to do your work. In today's lesson you will learn how to make new performers and give them different shapes and colours. You will also see how easy it is to make a performer change from one shape to another - this is known as morphing.

The tool bar down the left hand side of the screen is the main tool bar.



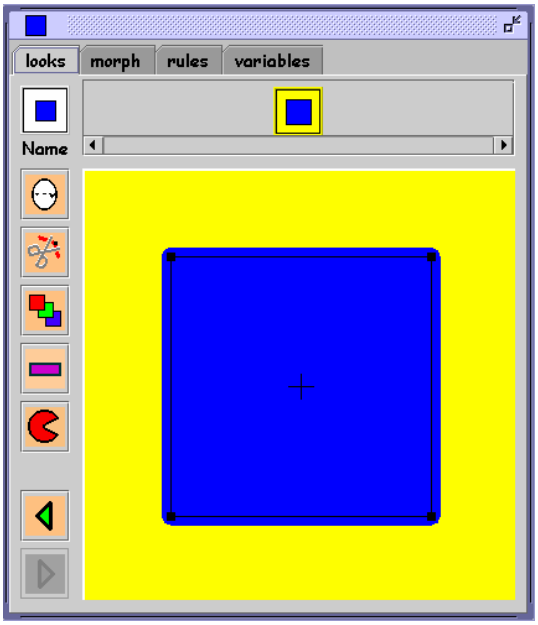
- Click the "create a new performer" button. The window that pops up is known as the performer's suitcase. Everything important about the performer is stored in the suitcase.



In today's session we will be working with two of the suitcase panels: the looks and morph panels.

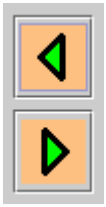
The 4 shapes are the starting shapes for your performers. It doesn't matter which shape you choose as they can all be changed to show any shape you want.

Let's start by clicking on the square shape. Straight away you see that the chosen shape fills up the performer's suitcase.



A square isn't the most interesting of shapes. It is very easy to change the square to something else. When you move the pointer over the square it changes shape. When it turns into a pointing hand you can grab the little boxes at the corner of the square.

- Grab a box and drag it around.



If you ever do something you want to undo you can click the undo button in the bottom left of the window. There is also a redo button that will redo the change.

Throughout Icicle there are lots of undo and redo buttons in case you ever make a mistake.

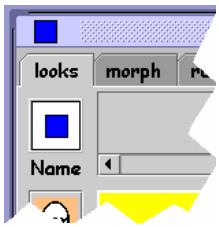
- Play around with making changes and then undoing and redoing. If you keep undoing you will go back to the shape selection panel, just press redo to get back to the square.

You can add more points to your shape by clicking near one of the edges of the shape. The new points are just like the original ones and can be dragged around as before.

You can also delete points by clicking on the delete button. When you have clicked on the delete button the pointer changes to a pair

of scissors. When you move the scissors over a point it will be highlighted and if you click then the point will go away. Remember you can always undo any actions with the undo button.

- Change your square into a boat. The changes you make are visible in two other areas of the suitcase. Can you find them?



The top left picture of the shape is special, it is known as the name shape and will become important later on.

If you want to change the colour of your shape you can press the colour change button. This gives you several different ways to change the colour of the shape.

- Choose a different colour. The undo button works at undoing this too.

There are some other buttons on the tool bar. Experiment with them.

What do these buttons do?

Let's make a new performer

It is a good idea to plan what you would like to do first. In your logbook draw a picture of a shape you would like to make e.g. a car, a house, a girl, a tree. Use the performer create button on the main tool bar.

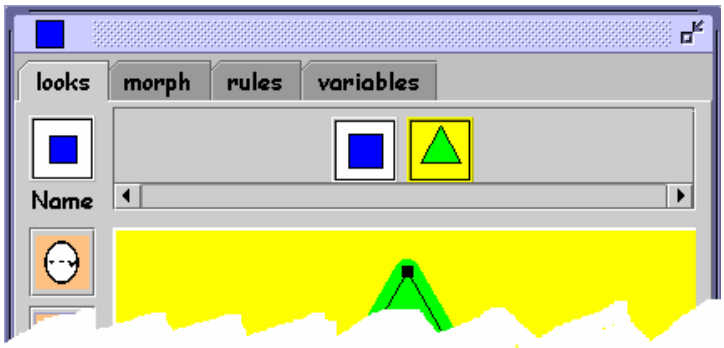
Choose the start shape closest to the shape you want to make. Then change the colour and make the shape look like you want it to.

Morphing

One of the things it is fun to do in Icicle is to get the performers changing from one shape to another. To do this the performer must have more than one shape in its suitcase.

There are two ways to create another shape in a suitcase. The first is how you made the first shape, click on the new shape button  . After you

have chosen the new starting shape you will see that there are now two shapes in the shape area of the suitcase.



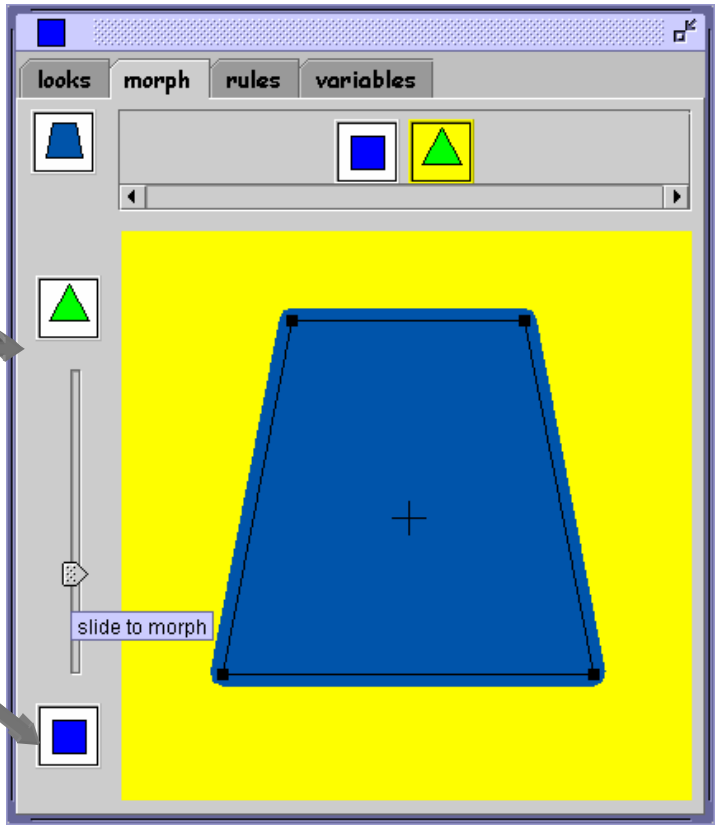
To work on one of the shapes you click on the desired shape in the shape area.

When you have two shapes you want to morph between you can select the morph panel of the suitcase.

On the morph panel there are two boxes on the left hand side to hold the shapes you want to morph between.

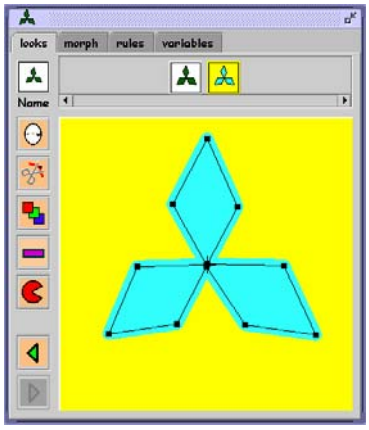
Drag the shape you want to morph fr… into the top box …

… and the shape you want to morph to into the bottom box.



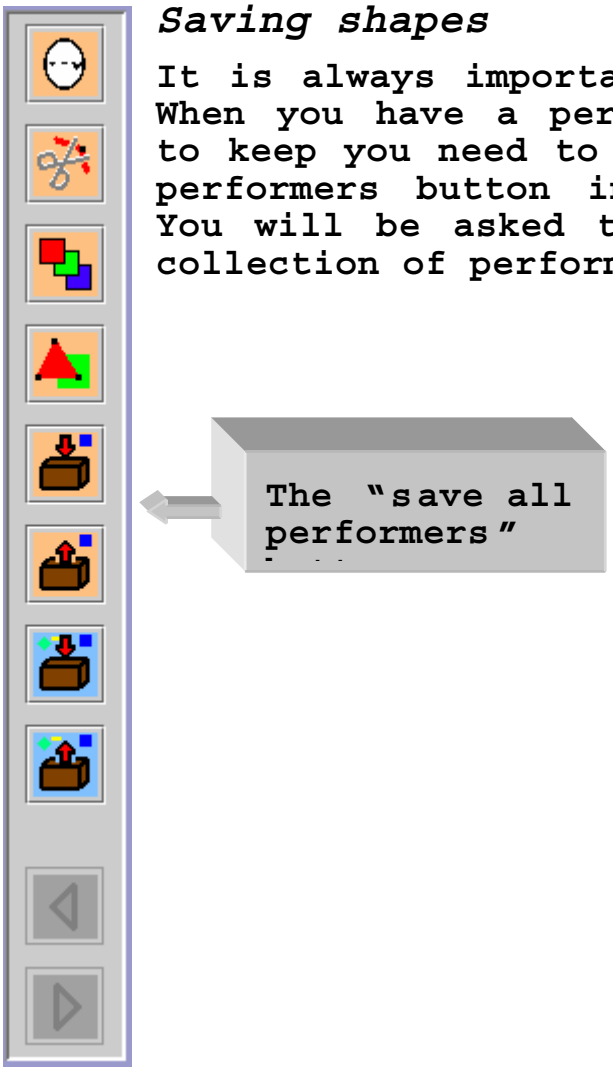
When you move the morph slider on the left hand side you will see the result of morphing between the two shapes.

Morphing not only changes the shape from one to another it also changes the colour. There is a simple way to change colour without changing the shape using the second method to create a new shape.



Go back to the looks panel of the suitcase. Drag one of the shapes and drop it on the shape area. This makes a copy of the shape. Copies are really useful when you want to morph a shape into a similar shape.

Now if you change the colour of the new shape you can try morping from the original to the new shape and only the colour changes.



Saving shapes

It is always important to save your work. When you have a performer you would like to keep you need to click on the save all performers button in the main tool bar. You will be asked to give a name to the collection of performers.

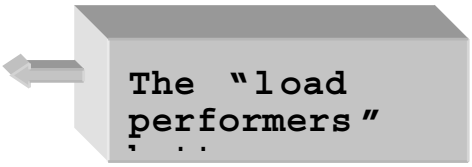
Today you have learnt how to make performers in Icicle and how to morph between two different shapes for the same performer.

Next time we will make our performers move around the screen.

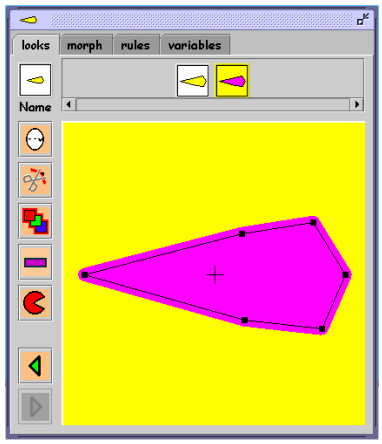
2.Making performers move

Today we start making our performers move about the screen. You can put performers on the screen by dragging a shape from a suitcase.

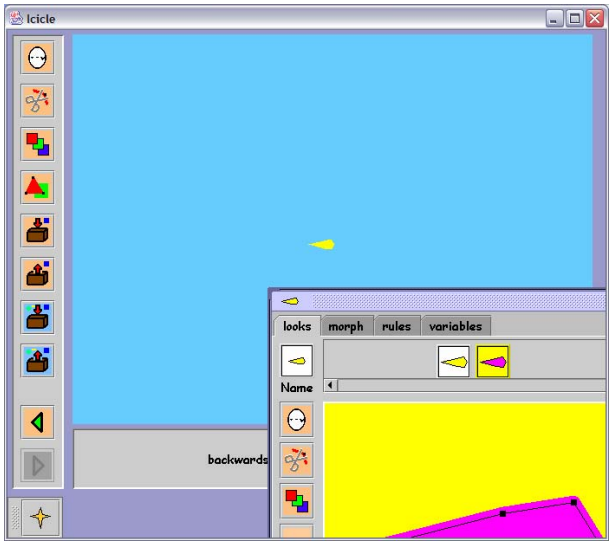
Load in your performers from last time. Use the "load performers" button.



You will see that the suitcases loaded in are closed. You can open a suitcase by



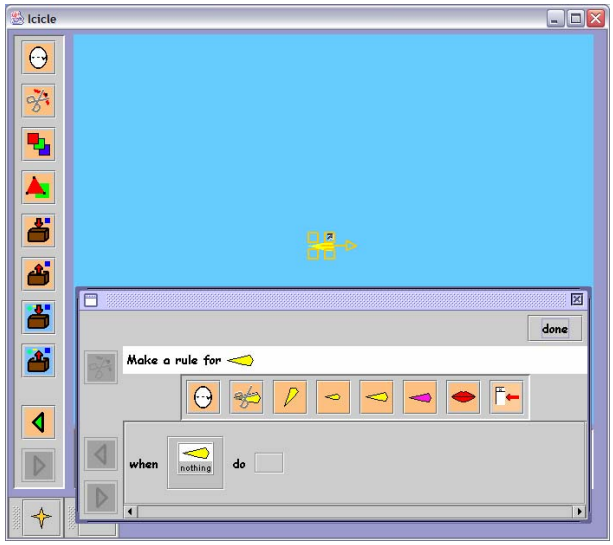
You can close an open suitcase by clicking in its "Minimize" box.



Open up a suitcase and drag one of the shapes from the shape area into the big blue area. The big blue area is known as the world.

Underneath the world there is a control panel – a little like a tape player control panel. To teach your performer how to do something you need to turn the world on. Press the Go button.

When the world goes it checks all of the performers in the world and tries to make them go. If it doesn't know how to make a performer go it asks you to teach it what to do. It does this by opening an editor window with a rule. Rules are the basis of everything that a performer can do.



The editor window will display "Make a rule for" followed by the name shape of the performer to make the rule for.

Every rule has a "when" part that shows when this rule will work. The first rule for every performer is the "when nothing" rule. This rule is the one that will work when there is nothing else for the performer to do.

You will see that the performer has four corner boxes and an arrow when the rule editor appears. We will look at these shortly.

- Click inside your performer and hold the button down (don't click inside the corner boxes at the moment).
- Drag your performer around the world keeping the mouse button down. What happens as you drag the performer?
- Let go the mouse button. What happens?

You will see some instructions appear in the rule editor.

There will probably be a turn instruction and a move instruction.

If you don't like where your performer finished up you can always use the undo button on the rule editor window and then drag the performer again.

All instructions in Icicle rules have little animations showing you what they do.

If you repeatedly drag the performer you will see more instructions in the rule editor window.

- When you are happy with the moves you have taught your performer click on the "done" button.

You have now finished your first program! Your performer will now be moving around the screen.

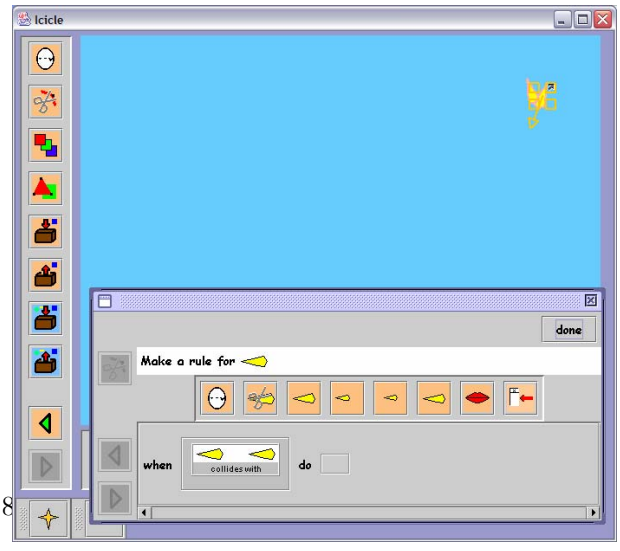
- See how the performer performs its instructions. It does the turn and then does the move and repeats this over and over again as there is nothing else to do.
- Drag another performer from the same suitcase on to the world and drop it somewhere away from the first performer. What happens?

All performers from the same suitcase have the same behaviour.

You can stop the world at any time by pressing the Stop button. You can also step through the program, either forwards or backwards by clicking the backwards and forwards buttons.

When a suitcase is closed there is a short-cut to drop a performer into the world. Drag the performer from the top of the small suitcase into the world.

- Add as many performers as you like to the world and make sure the world is going.



After a while two of the performers will collide with each other and Icicle will ask you to make a new rule for when this occurs.

- Move one of the performers

involved in the collision and then press "done".

Almost immediately Icicle will ask you for a rule when the performers "no longer touch".

- Just press "done", this means don't do anything. You will do this quite often when making an Icicle program.

Now, every time a pair of those performers collide they change course.

Collisions between different performers

Collisions can also occur between performers from different suitcases. When this happens the rule editor gives you two rules to make, one for each type of performer involved in the collision.

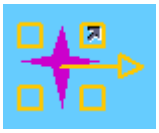
Deleting performers

Before we look at some of the other things a performer can do we need to get rid of the performers currently running around inside the world.

- Stop the world. (You don't really need to do this but it makes the next part much easier ☺.)
- Click on the "delete a performer" button in the main tool bar. The screen pointer will turn into a pair of scissors when it is over the world. When the scissors are over a performer the performer will be selected with its corner boxes and arrow. If you click now the performer will be deleted from the world.
- To stop deleting performers click on the world away from a performer.
- Delete all performers.

Remember: You can undo any of these deletions by clicking on the "undo change" button on the main tool bar.

- Once all of the performers are deleted from the world add a performer from a new suitcase.
- Select the performer, by clicking on it.



- **Play with the four corner boxes (handles) and the arrow. What do they do?**

You can use the handles when describing a rule as well.

- **Start the world going again.**
- **Make a simple rule to spin your new performer around without moving it.**

Remember the undo button.

An example

This example makes a bird fly around the screen and then disappear.

- **Delete all performers in the world.**
- **Make a bird performer**  **e.g.**
- **Drop the performer on the world.**

- When asked to make the "nothing" rule drag the bird around a few times then delete the bird from the world.
- Click "done".
- When the bird disappears drop a new one on the world.

Morphing performers

To make a performer morph its shape in a program you need at least two different shapes in its suitcase.

The way to teach a morph instruction is to drag the shape you want the performer to morph to from the suitcase and drop it on top of the performer whose rule is being learnt. The shapes must come from the same suitcase.

Editing a rule

We need a way to change rules after they have been described.

- Open the suitcase of the performer whose rule you want to change.
- Click on the rules panel of the suitcase.
- Click on the rule you want to change. This makes the rule editor panel appear again.

You can delete instructions from a rule by using the delete instruction button. It changes the mouse pointer to the pair of scissors. When you move the mouse pointer over an instruction it changes colour to show you that you can delete it. To stop deleting instructions click inside the rule area but away from an instruction.

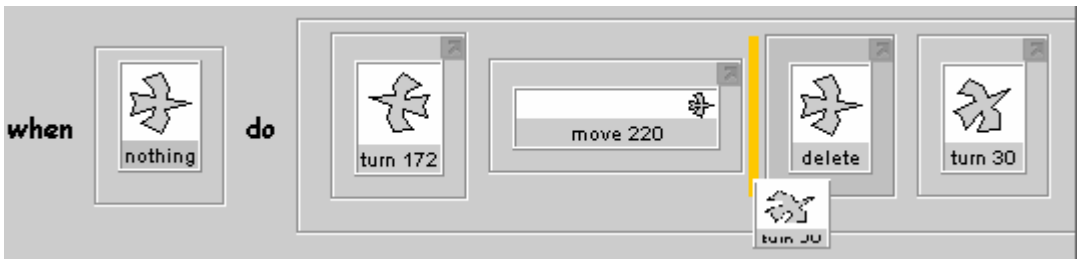
To add a new instruction you have to use the instruction buttons in the top row of the editor. Your instruction buttons will look different from the following because your performer shape will be showing on the buttons.

The buttons represent:

	create a performer (we see this in lesson 4)

	delete this performer
	turn this performer (starts by turning through 30°)
	move this performer (starts by moving 50 steps)
	zoom this performer (the same as dragging the corner handles, starts zooming by 150)
	morph this performer (only works if there are at least two shapes in the suitcase) – choose the shape you want the performer to morph to
	send a message to a performer (we see this in lesson 5)
	set a variable in this performer (we see this in lesson 6)

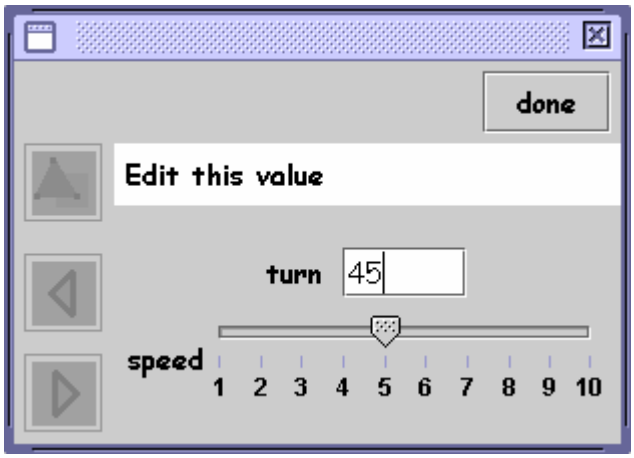
Clicking on one of these buttons inserts the corresponding instruction at the end of the rule. If you want to insert it somewhere else you drag the instruction from this position. An orange bar shows you where the instruction can be dropped. Once again the undo button can move it back to where it was.



Changing the instructions

When you use the buttons to insert instructions you commonly want to use a different value for the amount e.g. you may want the performer to turn by 45° rather than by 30°. Or you may want the performer to move by 10 steps rather than 50.

To make this type of change you right-click on the instruction you want to change – this opens an "edit this value" window.



You can then type in any value that you want. In some of the "edit this value" windows you can also change the speed the instruction will work at (1 is the slowest, 10 is the fastest).

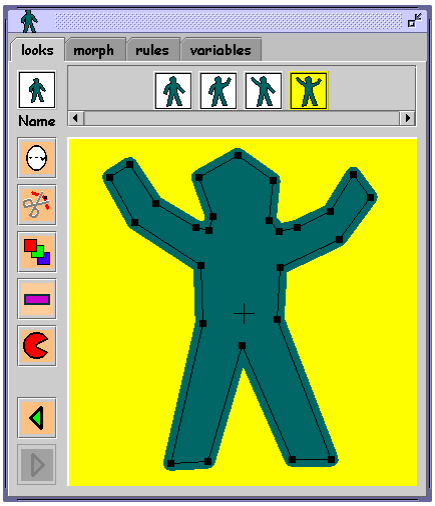
Today you have learnt how to make performers work automatically.

Next time we will make our performers react to the user.

3.Keyboard control and doing things in parallel

With the things you already know about Icicle you can make interesting patterns and make your performers do things when they collide with each other. Programs are more useful if they can be controlled by the user.

We will make a conductor performer and make it wave it's arms when the user types different letters.

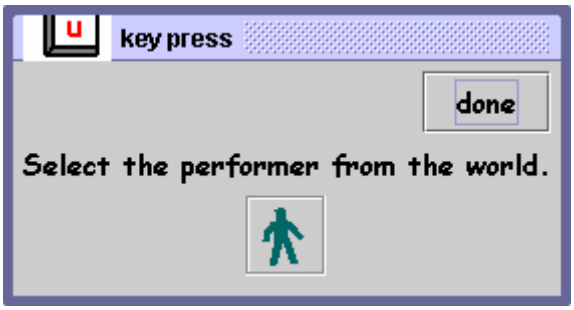


Here is an example conductor performer. This one has four shapes. You can start with two.

Play around with the morph panel to see the effect of morphing between your shapes.

- Drop a conductor into the world. If the world is off you might want to resize your conductor after doing this.
- Make the world go and when the rule editor window opens click on "done". This means that conductor will normally do nothing.
- Choose a letter, say "u" (for up).
- Press the letter "u" on the keyboard.

This opens the "select the performer" window.

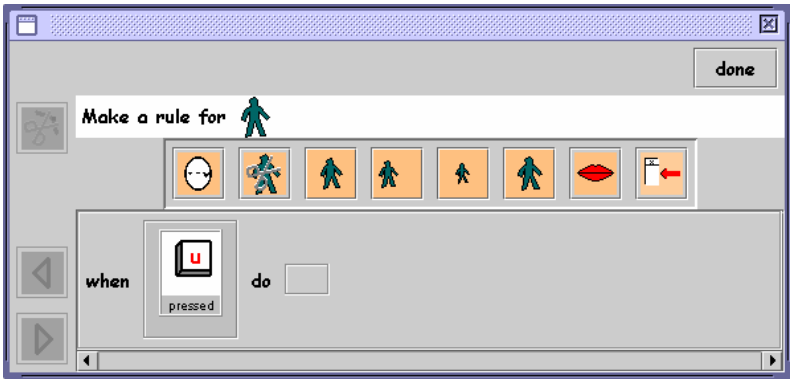


If your conductor was selected the box in the middle of the window will show the conductor. If your conductor wasn't selected you can now click on the conductor in the world and it will appear in the window.

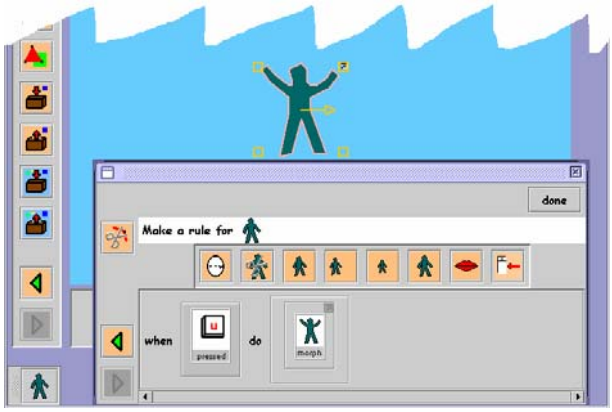
When you have the performer you want the key press to control in the window click "done".

If ever you type a key and the "select the performer" window appears by accident you can always safely get rid of it by clicking in the world away from any performers and then clicking "done".

You will then get a rule editor window wanting you to describe a rule for pressing the "u" key.



- Make the conductor morph from its current shape to another one. Remember there are two ways you can do this. You can drag the new shape from the suitcase and drop it on top of the conductor in the world or you can click on the morph button and select the shape you want the conductor to morph to.



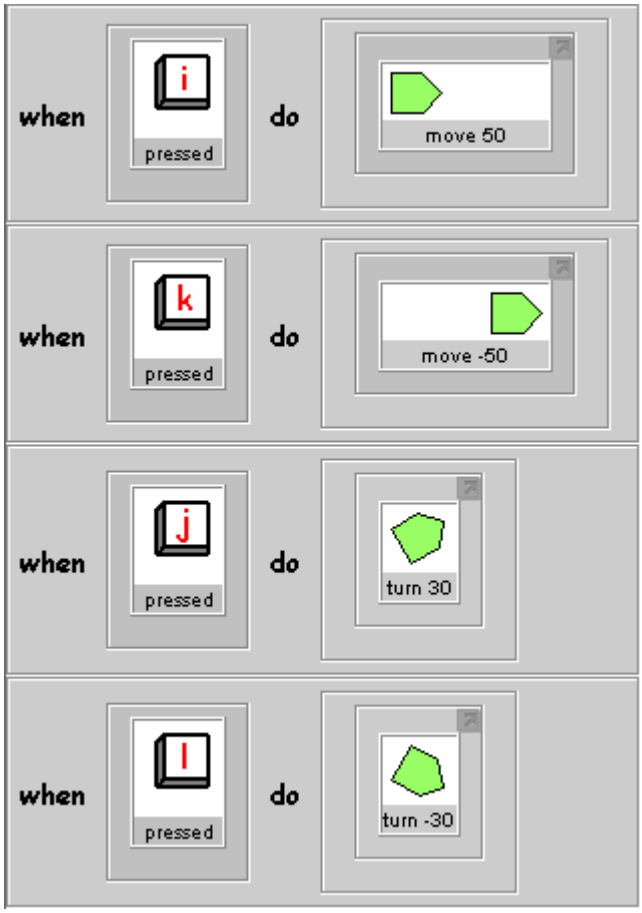
- Press "done".
- Press "u". The conductor will move its hands.
- Press "u" again. It appears that nothing happens. This is because the instruction is to morph to the shape the conductor already has and so you see no changes.

If you want the conductor to move its arms and then return to its original shape you will need to add an extra morph to the rule.

Now, every time "u" is pressed the conductor moves its arms up and then down. If you drop some more conductors into the world they will all move their arms together when you press "u".

Using keys to drive a performer around the world

Any instruction that a performer can perform can be included in a key press rule. A common use is to move the performer around the world.



With these rules a user can drive this performer around.

The "i" key moves the performer forward.

The "k" key moves the performer backwards. "move -50" moves the performer backwards by 50 steps. This was done by editing the value (see page 11).

The "j" key turns the performer to the left.

The "l" key turns the performer to the right. The value was edited to be -30.

- Make a performer and teach it how to drive around under your command.

Doing things in parallel

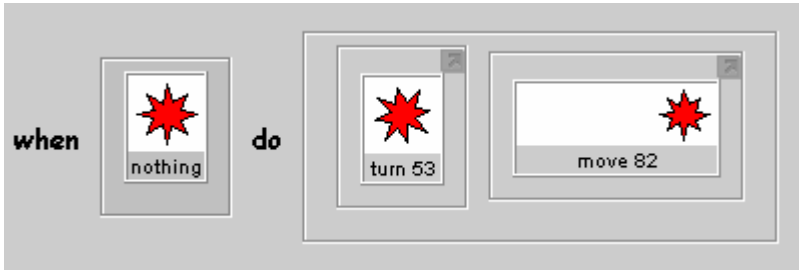
So far when one of our performers has moved it has turned and then moved. It would be nice if we could make a performer turn while it moved, making a nice curve as it goes around corners.

Icicle provides a simple way of combining instructions so that they happen at the same time. This is called "in parallel".

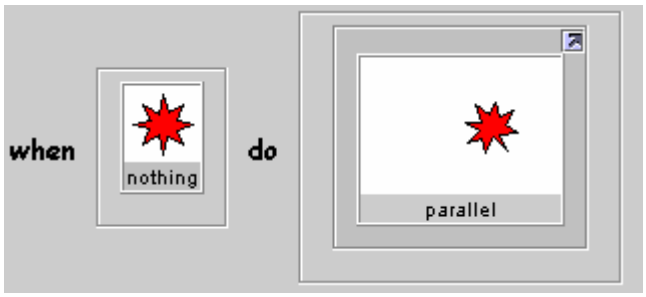
To see how this works:

- Make a prickly performer e.g. 
- Give the performer a "nothing" rule by dragging it across the world. Don't worry

about the actual values of the turn and the move.



- Drag the move instruction and drop it on top of the turn instruction (or the other way around). The rule will change to look something like this:

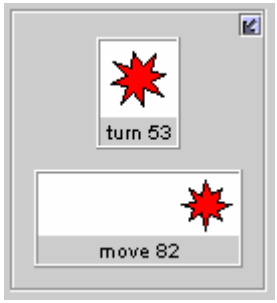


The instruction is a parallel instruction and the animation it shows has the performer moving in a smooth curve across the instruction box.

If you look closely you will see something a little different about the parallel instruction box.

- Click on the arrow in the top-right corner of the parallel box.

This shows you the instructions inside the box.



- Click again on the arrow and it goes back to the parallel animation.

You can have as many instructions in a parallel box as you like.

Another example

Here is a parallel instruction that makes our prickly performer zoom to a large size, while it morphs to a different shape and disappears. Useful if you want a spectacular way of getting rid of a performer.

Next time we will get our performers to make more performers while the world runs. This is useful to make things grow or to fire missiles.

4.Performers making performers

Opening a world full of performers.

In lesson 1 we saw how to save our performers. We usually want to save the world as well with the performers in their positions.

There are two buttons in the main tool bar to enable us to save and reload complete Icicle programs - all of the performers and where they are placed in the world as well as the world colour.

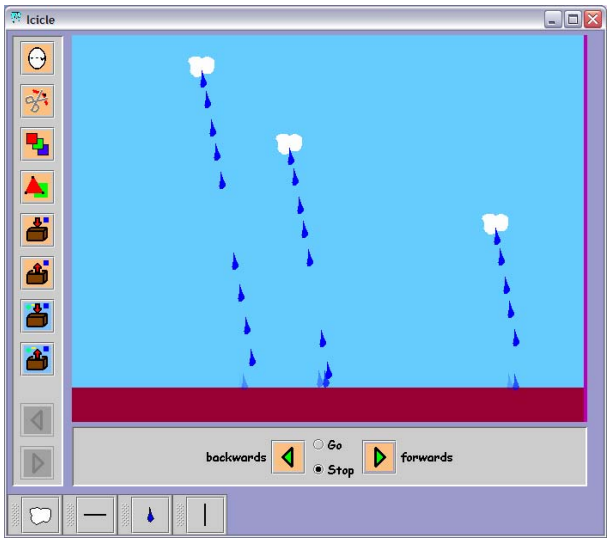
the save world button

the load world button

You have to be careful with the load world command because it will replace the current world and all of its performers with the new one.

Performers making performers

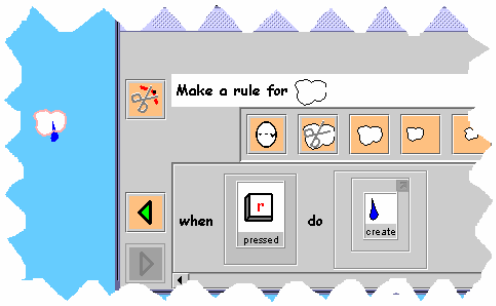
Many programs require new performers to be produced as they run. The picture below shows some clouds that produce rain when the user presses the "r" key.



As with most instructions in Icicle you can make a create instruction by dragging and dropping on the world.

The rule performer must be selected.

You drag the new performer onto the world and drop it. Where you drop it is important.



In this picture you see that the raindrop was dropped onto the cloud.

This means that every time the rule goes the new raindrop will be placed on top of the cloud.

The raindrop is placed relative to the cloud.

Try making a cloud that rains when the user presses the "r" key.

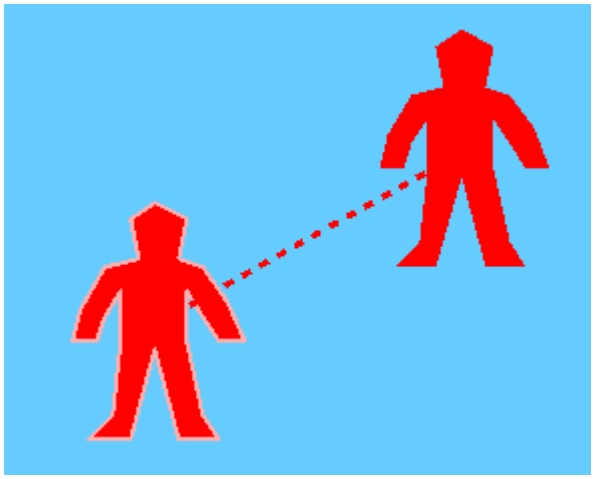
- Make a cloud performer. You can make it travel across the world if you like.
- Make a raindrop performer.
- Type "r" while the world is going and make the cloud create a raindrop.

When you make the world go again you will have to make lots of rules: the "nothing" rule for the raindrop, the "collides with" rule and the "no longer touches" rule.

- For the "nothing" rule drag the raindrop down to the bottom of the screen. You may want to delete it after it has fallen. What would happen if you didn't delete it?
- For the other rules just click "done".

You may see that your raindrop does not behave as you wanted. If you dragged it to the bottom of the screen the rule was taught as a turn and then a move. Sometimes we want our performers to move around the world without showing the turn.

Turning without changing the way the performer faces



When you are making a rule and want the performer to move in a particular direction without changing the direction it is facing you need to hold down the "shift" key as you drag the performer.

This picture shows a performer being dragged with the shift key held down. This is recorded as the following instructions:

The animation on the turn instruction does not show the performer turning it just shows an arrow turning. The arrow represents the direction the performer will move.

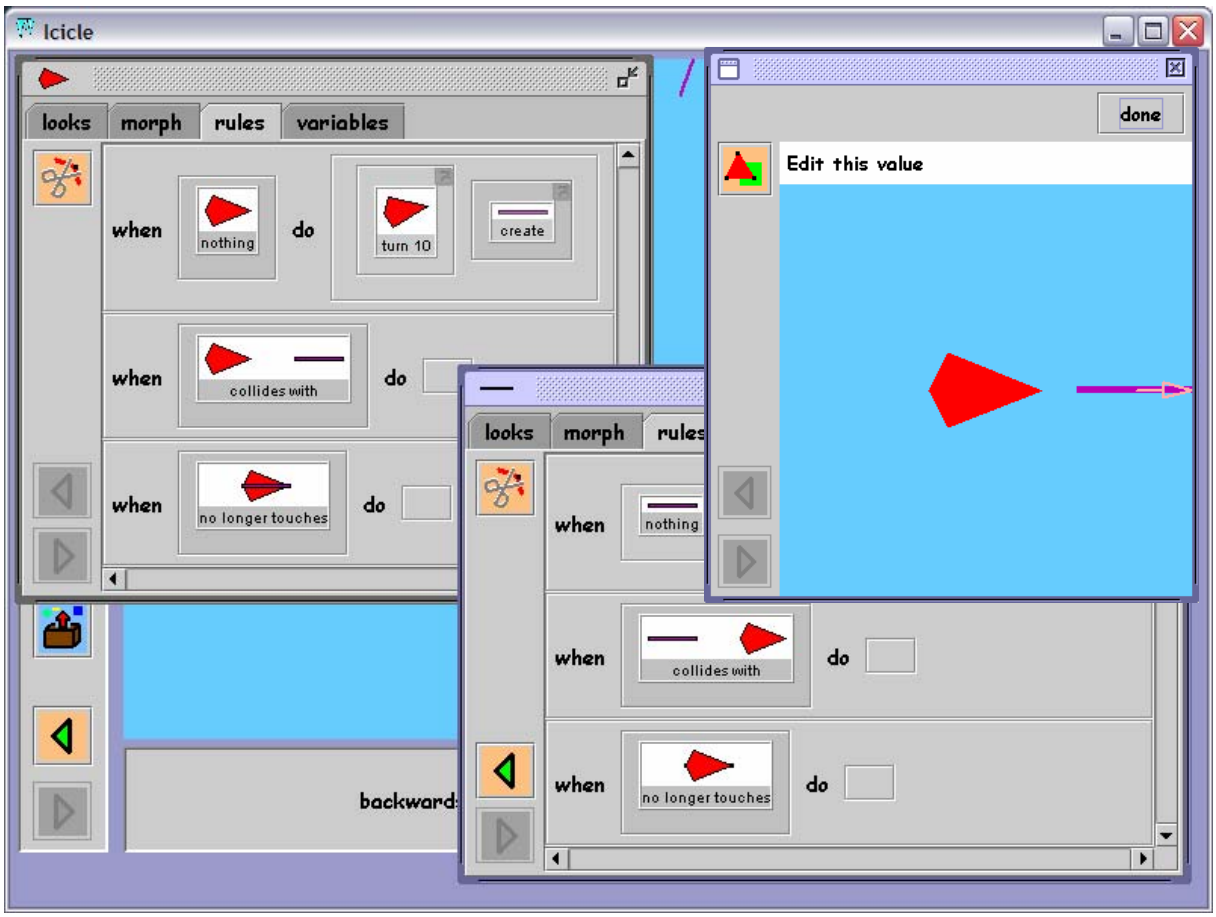
- You can use this approach to make the rain drops fall without turning them around.

Another example

We will make a shape turn around and around and fire rockets as it turns.

- Make a simple performer with a point on the right hand side e.g. 
- Make another performer of a straight line e.g. 
- Put the first performer in the world and make the world go.
- For its "nothing" rule make it turn a little bit and then create one of the straight line performers.
- For the straight line performers "nothing" rule make it move for a distance and then delete itself.
- You will have to deal with "collides with" and "no longer touches" rules.

You should end up with rules similar to these:



Creation editor

If the rockets aren't created in the correct position or facing the way you wanted you can alter things with the creation editor.

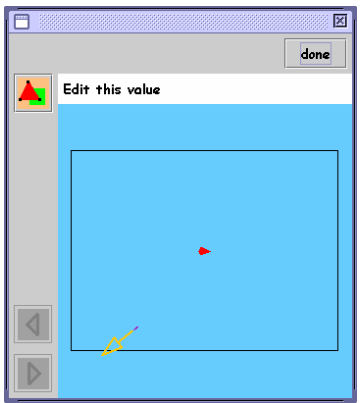
- Open the rule with the create instruction in it.
- Right-click on the create instruction.

You will get a window like this:

In the centre of the window is the performer that does the creation. The created performer will have an arrow showing the direction it will face when it is created.

You can drag the created performer around and put it in a new position and change the direction it will face by dragging the arrow.

If you drag the performer a long way away from the creator everything shrinks to keep the picture inside the window.



If you drag it too far away a rectangle appears in the window

showing the size of the world's display so you know how far away from the creator the new performer will appear.

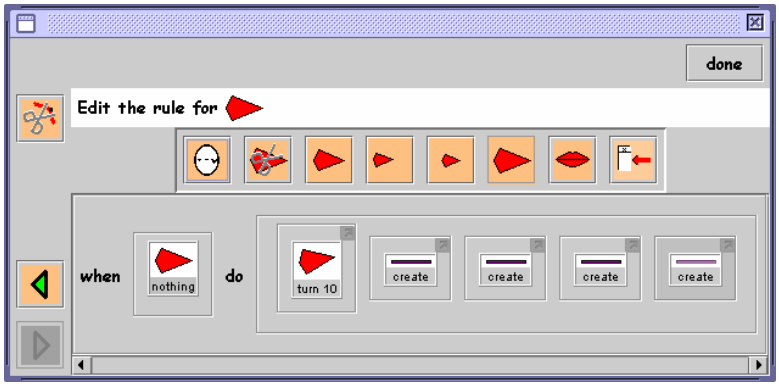
When you have the new performer in the place you want press "done".

Making in parallel

One of things you can do with the creation editor is to use one performer to set up a number of other performers. Doing this provides a simple way of positioning many performers at different distances and directions.

- Open the rule with the creation instruction.
- Create four more rockets by pressing on the "create a new performer" button  in the rule editor's toolbar and selecting the line shape.

You should have a rule that looks like this:

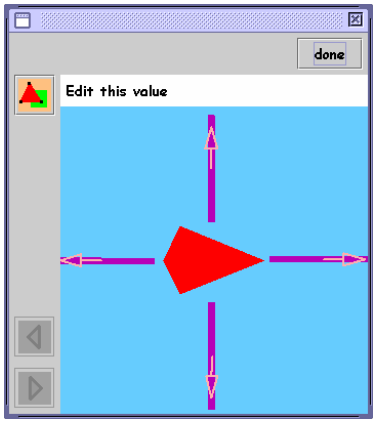


- Drag the four create rules so that they are all in parallel.

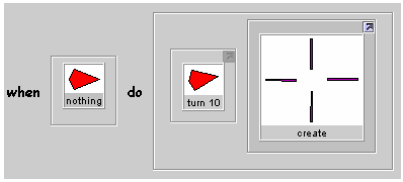
It looks as though you now have only one create instruction.

- Open the creation editor on the create instruction.

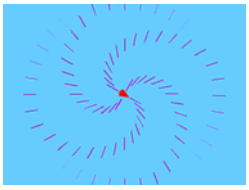
You can now drag all four created performers and place them as you would like.



When "done" is pressed the create instruction shows how all four performers will appear when created.



- Now run the program.



Too many performers

You have to be very careful when creating performers especially if you do it inside the "nothing" rule because this goes very frequently. There is a limit on the number of performers the Icicle world can contain.

We have seen how to create performers from within our rules. Next time we see how to send messages from one performer to another.

5.Sending messages

Sometimes our performers need to do things because of something done by another performer. Icicle allows us to do this by letting performers send messages to other performers.
Messages in Icicle are called star messages because they are always shown as a star. e.g.

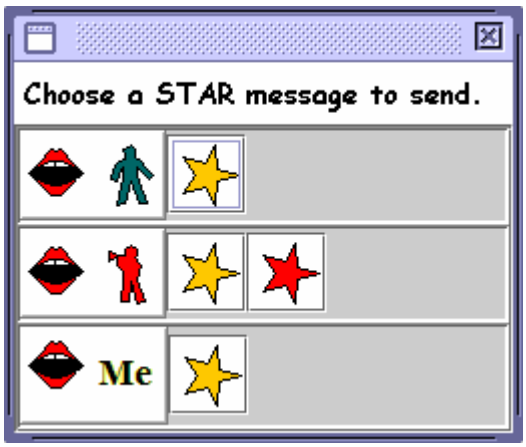
The colours of the stars change and the number of points in the stars change to give different names to different messages. e.g. a five pointed green star would represent the message "green 5", a seven pointed red star represents the message "red 7".

When a message is sent to a performer it causes an interaction just like a collision or a key press. So every message has its own rule with its own set of instructions.

A good example of message sending is an orchestra program. When the conductor sends a message to the trumpet players they move. When the conductor sends a message to the violinists they move. It doesn't matter how many trumpet players or violinists there are. All performers from the same suitcase get the message at the same time.

The only way to teach a message instruction is with the send a message button .

When you press this button a window appears asking you to choose a message to send to a performer.



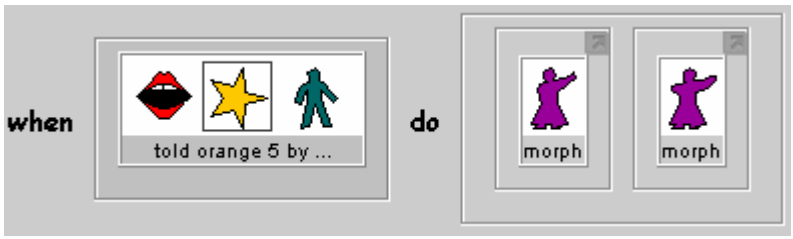
There is one row for each type of performer in the program. Each performer has a number of star messages you can possibly send to it. You click on the (star) message you want to send. In the example above the trumpeter has two possible

messages because it already has a rule for the red 5 message. So you can either send the same message or send a new message.

Message instructions look like this:

This sends the "orange 5" message to all violinists.

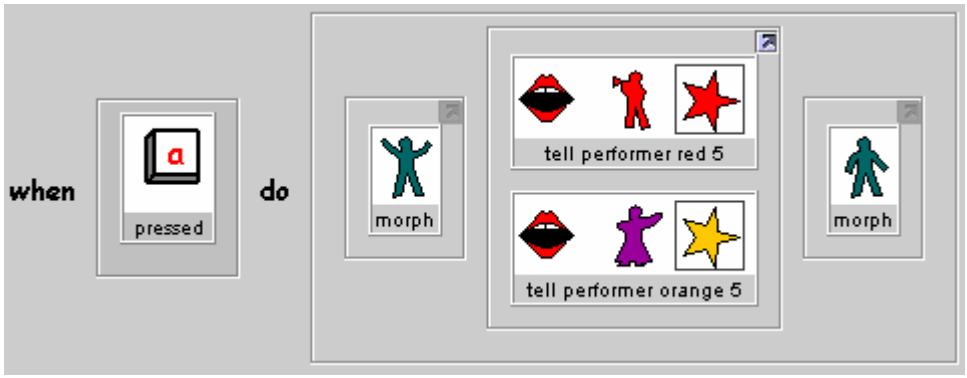
When this message is received you have to make the rule for "told orange 5".



The "told orange 5 by …" condition includes a picture of the performer that sent the message.

This rule means that the violinists will play when the conductor sends the "orange 5" message.

If you want to make the violinists and trumpeters both play at the same time you can do it with a conductor rule like this:



In this case when the user presses the "a" key the conductor raises his arms, then the violinists and trumpeters get sent messages in parallel and then the conductor lowers his arms.

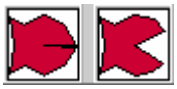
When the violinists and trumpeters get their messages they play their instruments.

Following another performer

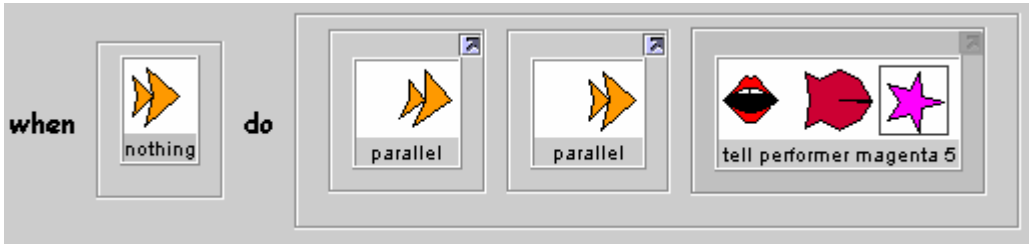
When a message is received from a performer the receiving performer has a new type of instruction it can run - the "turn to this performer" instruction. It is represented by a button showing both performers (the sender and the

receiver). The receiver turns to face the sender. This can be used to make one performer chase another.

- **Make a fish performer and give it a couple of shapes**  **.**
- **Make a shark performer and give it a couple of shapes with its mouth open and closed.**

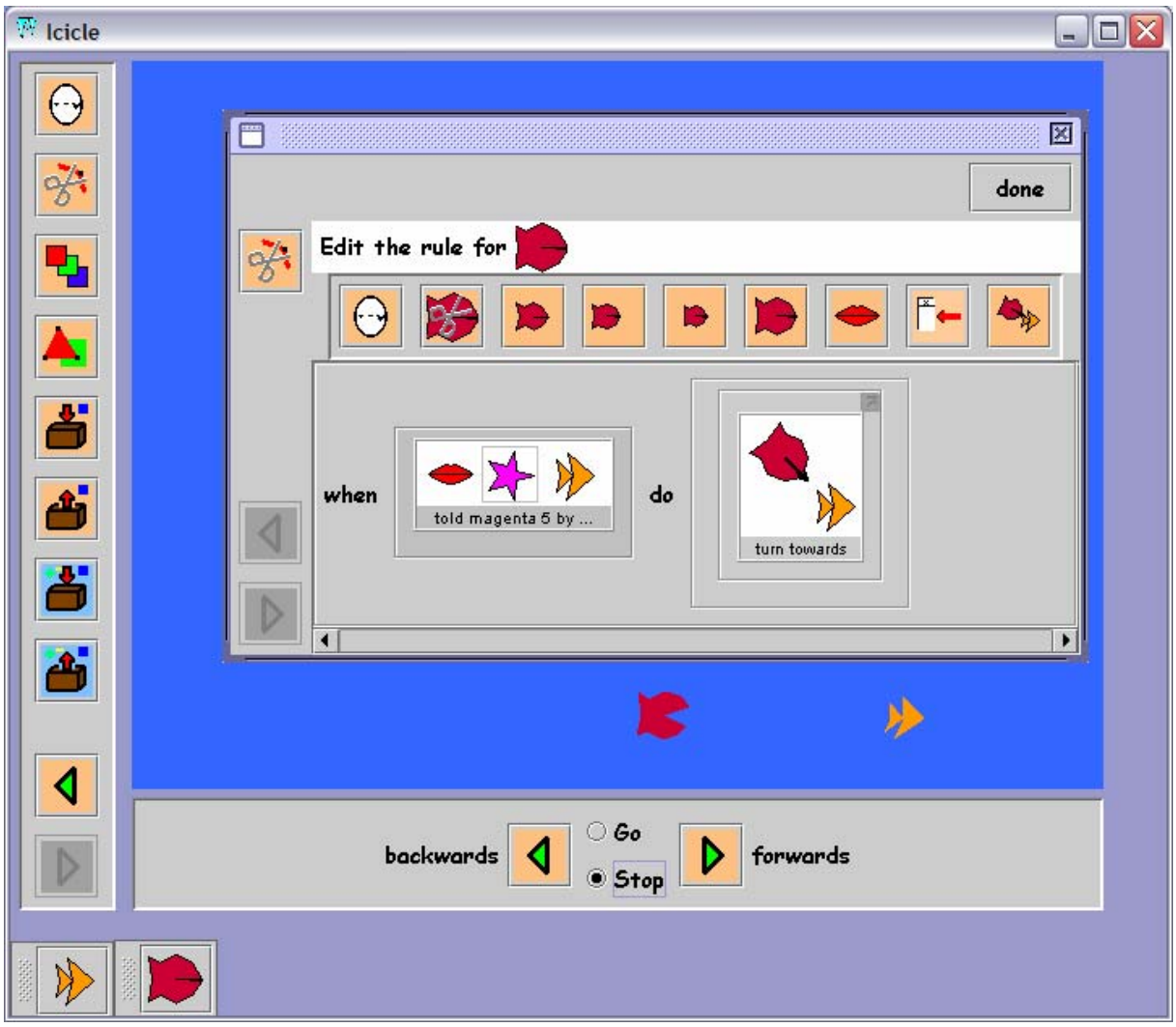


- **Make the fish swim across the screen and send a message to the shark.**



- **Make the shark swim across the screen opening and closing its mouth.**

- **When the shark gets the message from the fish, make the shark turn to the fish.**



- Run the program.

It must be a pretty stupid fish if it keeps sending messages to the shark telling it where it is.

If you drop a lot of sharks and fishes into the world you will see that all of the sharks chase one fish at a time. You will also need to write a rule for what happens when a shark catches a fish (or bangs into another shark).

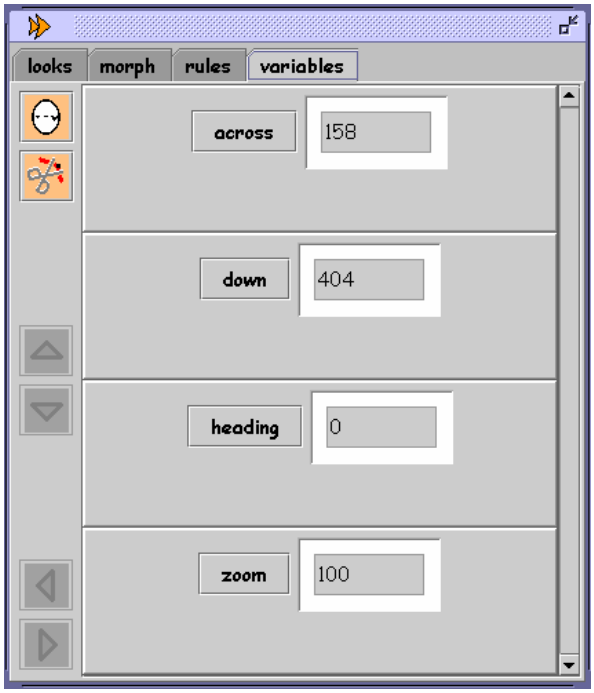
In this lesson we saw how to send messages from one performer to a group of others. Next time we will see how to do some simple arithmetic with our performers and keep scores for a game.

6. Using variables

Sometimes we need to keep track of numbers in our programs. For example you might want to keep a count of how many treasure chests your performer has collected. The game might finish when all have been collected.

Any sort of counting or scoring can be done easily in Icicle using variables. A variable is like a box that contains a number. The program can use the value stored in the box. When a new value is stored in the box it replaces the value that was there previously.

All variables are visible on the variables panel of the suitcase.



There are four standard variables for every performer. These cannot be deleted.

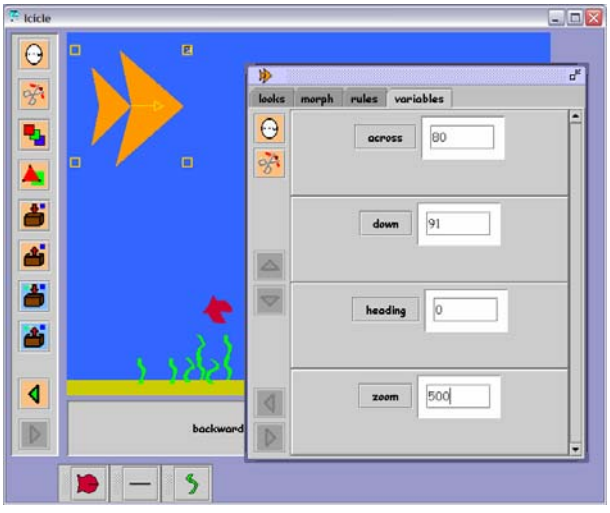
across: how far across the world the performer is

down: how far down the world the performer is

heading: the direction the performer would move (0 is to the right, 90 is straight up)

zoom: how big the performer is (100 is normal size, 200 is 2 x normal size, 50 is ½ normal size)

If you select a performer in the world the suitcase it came from will show its variable values. It also allows you to type a new value in. In this picture you see a fish that has been zoomed to 5 times its usual size



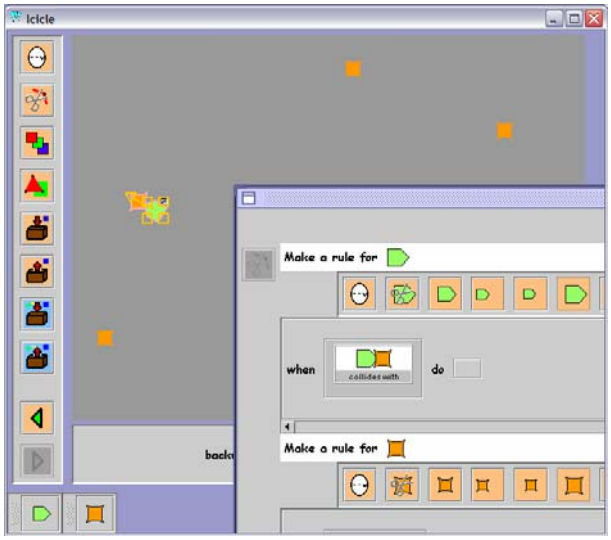
because 500 was typed into its zoom variable.

You have to press the enter key after typing the value to make it happen.

Keeping score

We will see how to keep a score for a simple performer moving around the world collecting treasures. The moving performer is the car from lesson 3 that can be moved around the screen by the user typing the "i", "j", "k" and "l" keys.

- Load the "driving" world.
- Make a treasure performer e.g.  .
- Drop several of the treasure performers on to the world.
- Start the world and give the treasure performers an empty "nothing" rule.
- Drive the car around until it collides with a treasure.



- For the treasure performer's rule delete the treasure.

For the car performer's rule you need to make a new variable called "treasures".

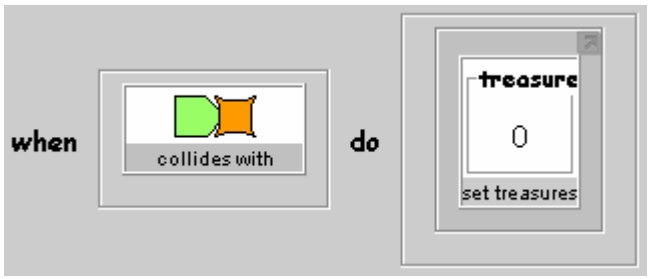
- Open the variable panel of the car's suitcase.
- Click on the "make a new variable" button  .

- Type the name "treasures" in the window that pops up and then click "done".

The new variable will appear on the variables panel with a value of 0.

- Now go back to the "Make a rule" window and click on the "set a variable" button for the car. This opens a pop-up list of all of the performer's variables.
- Click on the "treasures" item in the list.

The rule should now look like this:

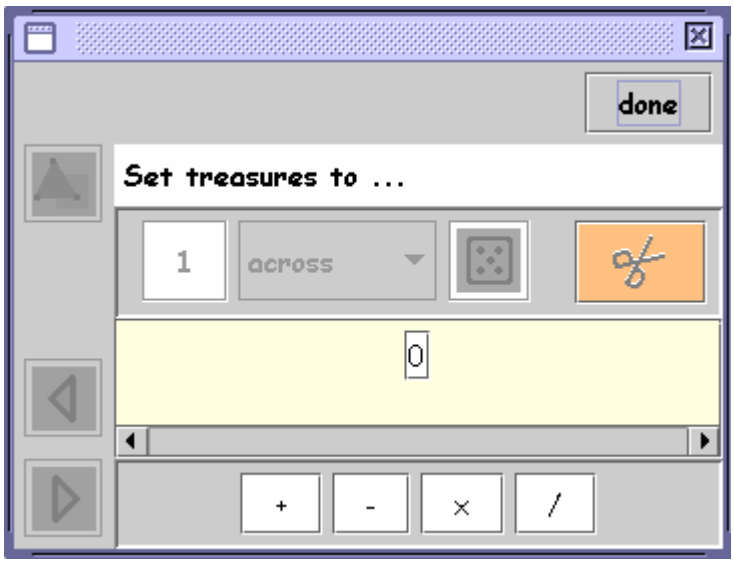


The instruction with the word "treasure" at the top is the instruction to put the value 0 into the treasures variable.

Unfortunately we don't want to put 0 into the treasures variable we want to add 1 to the value already there. Remember that this rule will work whenever a car performer collides with a treasure performer and we want the treasures variable to count how many treasures the car has collected.

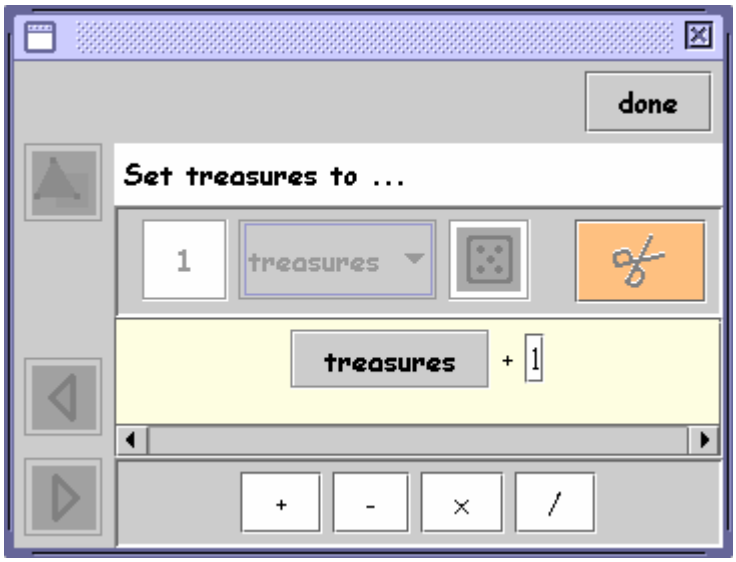
- Edit the "set treasures" instruction by right-clicking on it.

The set variable editor window opens. This window is quite complicated but what we want to do is to set the treasures variable to be the old value plus 1 more. This is represented as "treasures + 1".

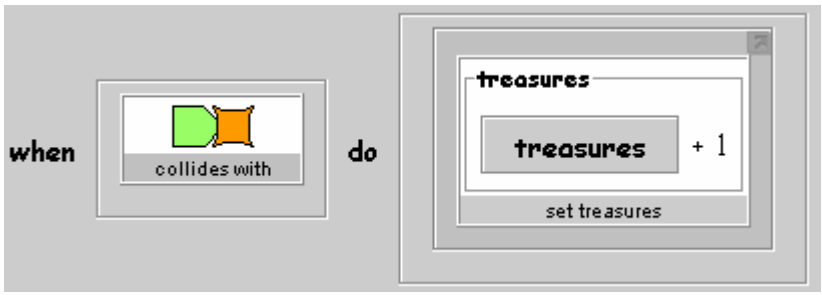


- Click on the "remove last" button  . This gets rid of the 0 in the centre.
- Click on the pull-down list of variable names and choose "treasures".
- Click the "+" button.
- Click the "1" button.

The window should now look like this:

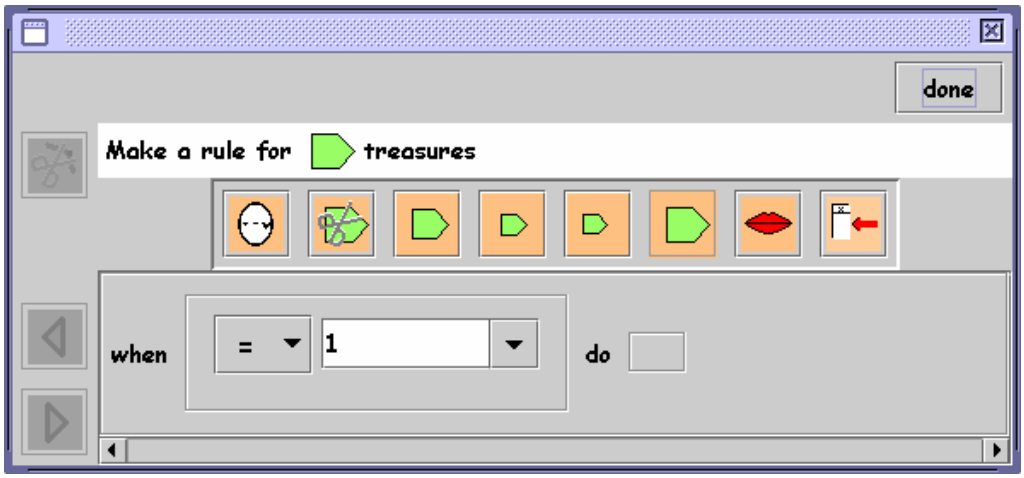


- Click the "done" button and the rule now looks like this:



This means that when the performers collide the "treasures" variable will have its value go up by 1.

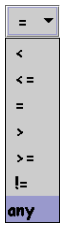
Now when the program runs the value of 1 gets stored in the treasures variable. This makes Icicle ask for a rule just like for any other event. So setting a variable is just like a collision, a key press or receiving a message.

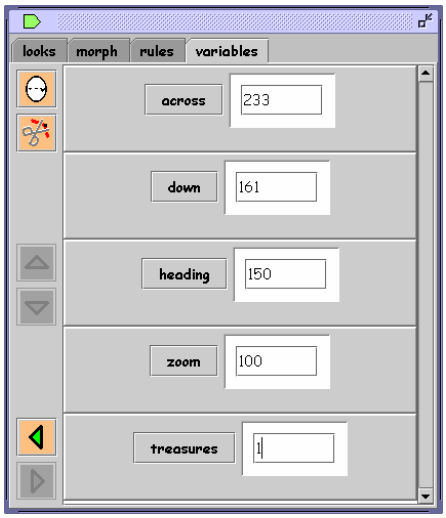


The difference is that the condition can be changed. In this case we want the same rule for

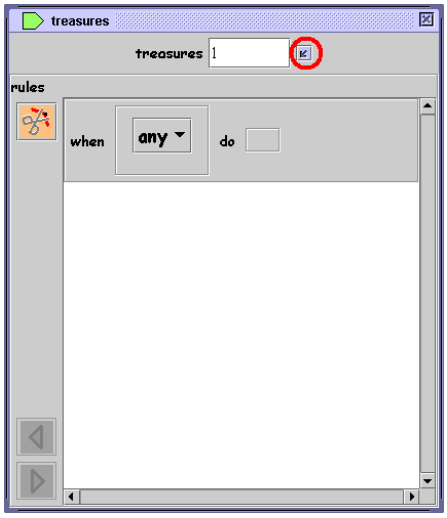
any value being assigned to the treasures variable.

- Click on the "=" sign and select "any".
- Then click on "done". This means that for any value being put in the "treasures" variable nothing will happen.





- Now open the variables panel of the car's suitcase.
- Click on the button that says "treasures". This opens the "treasures" variable rule page.



Quite often we just want to see the value of a variable.

- Click on the little arrow, it is circled in the picture.

This shrinks the variable rule page to just show the value. This is now small enough to leave on the screen and as you drive around the screen hitting treasures you can see how many you have collected.

There are all sorts of other things you can do with variables, they are the most powerful part of Icicle. Here are some examples:

Counting

If you want a performer to change its behaviour over time you can do this with a variable. When

the value of the variable gets to the number you want you can make a new rule for it.

Teleporting

You can also use variables to make performers jump around the world. In fact that is exactly how the performers jump to the other side of the world when they move off an edge. You can see how this is done by looking at the rules for the "across" and "down" variables.

This is the end of the Icicle tutorial, there are still other things that Icicle can do. In the next six sessions we will be making programs with Icicle including ones you want to make.

APPENDIX B

This appendix collects the changes made to Icicle during the testing process. Changes were made because some of the original approaches were difficult or because the children wanted extra capabilities added to Icicle. In most cases, the reason for the change is described, as well as the action taken. In a few cases, a recommendation is made for future improvements.

In an attempt to provide some order to these suggestions they have been grouped together into sections that corresponded to the six sessions in the tutorial booklet (Appendix A).

Session 1 - The production of performers and shapes and using the morph panel

There were several minor difficulties with the production of shapes.

The scissors tool

The scissors tool to delete a point was designed to stay active until the user clicked on the editor window but away from a point. This way multiple points could be deleted and the tool deactivated without having to return to the toolbar. The children said they expected to be able to turn the tool off by clicking on the scissor button again. I changed the tool to deactivate using either technique.

Off-centre shapes

A more important difficulty was that some of the children did not create their shapes centred on the cross hairs in the middle of the editor. I was surprised by this as the initial shapes the children had to start with were all centred correctly. The children had to explicitly drag, add and delete points to produce shapes away from the centre (Figure 112).

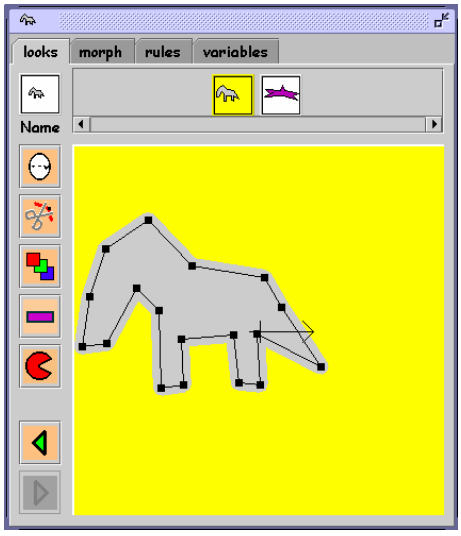


Figure 112 – An off-centre elephant shape.

This was a difficulty because the cross hairs determine the centre, and hence position co-ordinates, of the performer when it is placed in the world. All turn and zoom instructions are centred on this point. When a shape is created in the editor there are controls to rotate the figure 90° at a time and to

reflect in the vertical axis through the centre of the editor but there is no way to translate the shape. A technique to do this, e.g., by grabbing the shape, needs to be added.

Shapes facing the wrong way

A related difficulty was that several of the shapes were drawn facing the left or the top of the shape editor, such as the elephant in Figure 112. When given a move instruction, with a positive distance, a performer will move to the right as it appears in the shape editor. This meant that extra help had to be given to the children to show them the way to orient their performers in the editor. Figure 112 shows the arrow that was added to the editor pointing to the right from the centre cross hairs. This is still not satisfactory as it necessitates children drawing performers such as rockets on their sides. At the moment, children can draw such performers upright and then rotate them in the editor to head towards the right but it would be better if the facing arrow could be turned to indicate the intended forward direction of the shape.

Partially morphed performers

As the study progressed, the children frequently gave performers morph instructions and when they used the morph panel to examine how a morph would appear they found they liked one of the shapes partway through the transition and wanted to use that as a shape for the performer. Icicle currently only allows shapes from the shape list to be dropped into the world. A simple addition would allow transitional shapes to be dragged into the shape list.

Undo not used

One aspect of working with the shape editor that surprised me was that the children did not use undo very often. Instead, if they wanted to return a shape to what it was they modified it by adding, deleting and dragging points. I repeatedly pointed out the undo facility but this was usually ignored. Similarly, when the children were constructing their worlds they seldom stepped backwards to inspect what had happened.

Paper shapes better

As the best shapes were produced when I asked the children to draw what they wanted on paper first, it appears that some more work needs to be done on making the shape editor an effective way of producing performer shapes. The interaction with the shape editor feels more like constructing a shape than producing a drawing.

Where has the suitcase gone?

The name given to the window that contained all of the information about a performer type changed from "suitcase" to "performer box" after the study. The original term of "suitcase", used in Appendix A, was chosen to signify holding all of the material associated with a performer type. The early design for the onscreen representation even had handles and locks on the suitcase. This was to be part of an Icicle user-interface style that was abandoned due to considerations of complexity and time. The name was kept in the tutorial booklet but it was not represented in the interface and the children seemed puzzled by it.

Session 2 - Dealing with interactions, giving performers instructions and editing instructions

Dragging performers into the world

The only ways a performer could be dragged into the world were by dragging from the shape list (Figure 34) or from the icon in a minimized performer box (Figure 36). As the children frequently attempted to drag from the name shape box, I made this work as well.

Irritating rules

As mentioned in Chapter 4 the children really did not like being interrupted unnecessarily for interaction rules. In particular the children never used "no longer touching" rules during the study. Therefore, when these interactions occur, an empty rule is automatically created and the user is not interrupted. If users do want to add instructions to the empty rules they edit them as usual.

Zooming

Zoom instructions were very popular – large effect for little effort. Unfortunately if a zoom instruction is repeated, the performers get bigger and bigger. Problems occur when performers are larger than the world as nothing else is visible. The opposite is also a problem when the performers get too small to see. When a child made a performer grow in the "nothing" rule they did not always want the performer to continue to grow. They wanted the performer to grow to a certain size and then stop. A pair of girls produced a growing camel performer but they solved the size problem by making it shrink when it hit another performer.

Small controls and small performers

The children commented on the size of the rotate, and enlarge orbiting controls that become visible when a performer is selected. They wanted the controls larger, and even though they did not comment on the default size of the performers (I presume because they can change the size easily), they did find it difficult to select small performers, especially skinny ones. In order to select a performer the pointer has to be inside the area of the performer before the mouse is pressed. A more forgiving selection technique should be provided.

Bring to top

Another instruction that was added as a result of testing was the "bring to top" instruction. Some programs required performers to remain on top of other performers even when the other performers had been added to the world later; adding a performer to the world always puts it on top of the other performers unless the user explicitly moves it to the bottom. The "bring to top" instruction solved that problem but is only a temporary fix. A more general solution could use a variable for each performer representing the layer the performer is currently in and collisions would only occur between performers in the same layer. By default all created performers would be in the same layer as their creators.

Session 3 - Keyboard interactions and making parallel instructions

Arrow keys

Originally, the Icicle key press handler did not recognise the arrow keys. If a performer was to be controlled by the keys the children had to use ordinary alphanumeric or symbol keys. The children really did not like this and requested the ability to use the arrow keys for direction. Arrow keys now produce interactions showing a key being depressed with the corresponding arrow on the key top.

Unwanted key presses

Key press interactions were very easy to produce, quite often too easy. As soon as a key is pressed when the world is running the system checks to see if there is a performer that responds to that key press. This means that it was very common for children to be playing a game and to press the wrong key by mistake. As there was no performer that responded to that interaction, the game was stopped and the user was asked to pick a performer and define a rule. This highlights a wider problem, that of asking for rules whenever particular interactions occur. There should be a way of turning off the learning of interactions, especially when a program is going to be used by someone apart from its programmer.

Session 4 - Creating performers and the creation editor

Improved create animation

In Appendix A the animation for the create instruction displays only the created performer. Before the children got that far through the material, I had changed it to show the creator as well. This works much better and is a closer match to the creation editor when the instruction is edited.

Large numbers of performers

As expected the children used the create performer instruction to produce large numbers of performers. Unfortunately the more performers the slower Icicle runs (as mentioned in Chapter 6), so an arbitrary limit on the number of performers was imposed – initially one hundred. This was not an ideal solution.

The large number of performers' problem, along with the problem of using the zoom instruction to produce incredibly large performers, arises because of the ease of producing the effect. In practice, after constructing a few worlds with these effects, the children either got bored with them or frustrated at the speed of Icicle when there were a large number of performers. More work needs to be done to improve Icicle's performance with large numbers of performers.

Improved creation editor

Originally the creation editor could not resize created performers. The created performers were made to the scale of the creator performer. Unfortunately this meant the performers had to be designed to scale with each other. This required preplanning by the user. The children did not do this and so the creation editor now allows a created performer to be sized relative to the creator. This is much nicer and fits with using relative values throughout the Icicle instruction set.

Session 5 - Message sending and turning to face other performers

Before the children reached the section of the tutorial dealing with messages, the screen representation of potential messages had changed from that shown in the tutorial. In Appendix A, a separate window opens to display the messages. The version the children used displayed a pop-down list (see Figure 58) in the same way as the "create", "morph" and "set a variable" instructions.

Session 6 - Variables and simple variable interactions

Default assignment rule condition

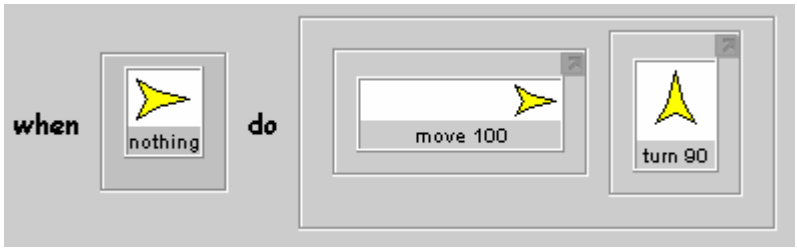
Variable assignments originally produced rules with the condition "when = value" when a new value was assigned to a variable. If the user wanted to generalize the condition, they had to do this explicitly. This meant that users received requests for rules to be defined every time a different value was assigned to a variable. As the children usually did not generalize the condition, even though that was the result they wanted, this became a source of frustration. To reduce this frustration the default condition for a variable was changed to "any". If a more specific rule condition is required, such as "when >= 10" the user has to explicitly alter the condition.

Entering variable values

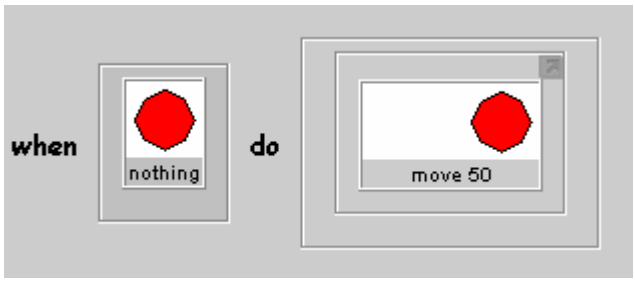
When the children were shown how to produce rules to do things like repeatedly adding one or multiplying by two there was a slight user interface problem. When a value was typed into a variable box the child had to press the enter key to signify they had finished typing the number. Very frequently the children did not do this. They typed a number and then continued working on some other part of the program. The box would display the number but the value actually stored in the variable was not altered, as it had not been entered. This added to the difficulty the children had with variables. It was not possible to store the number into the variable as the child entered it because if they wanted to enter 256 they did not want to execute the rules corresponding to 2 and 25 before the rule for 256. This was fixed by entering the value into the variable either as soon as the keyboard focus was lost by the box or when the enter key was pressed.

APPENDIX C

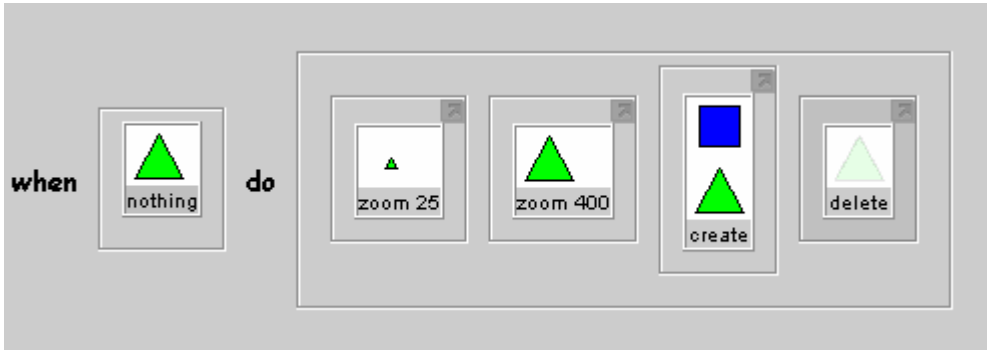
Static images used to test the understanding of Icicle rules. The children were asked to describe when the rule ran and what it did.

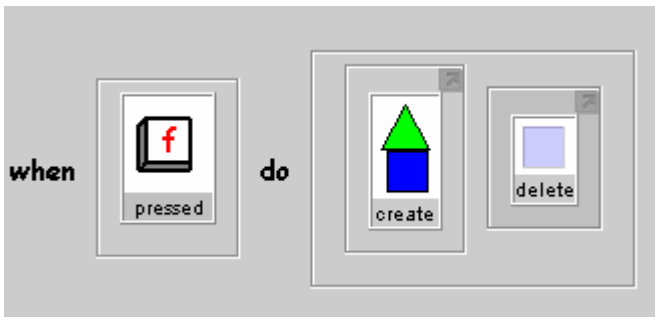


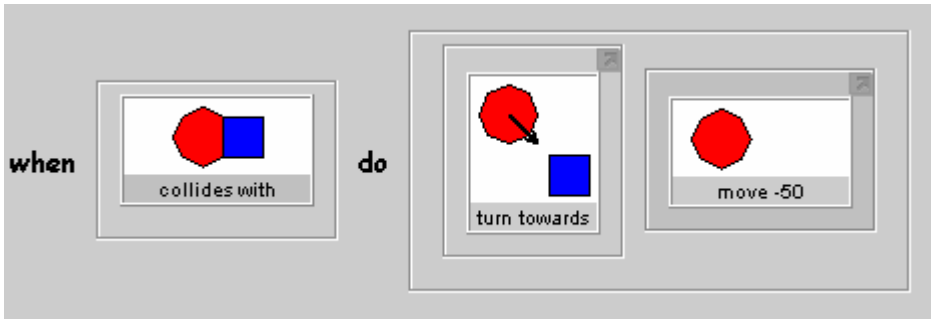
What pattern does the performer make as it moves on the screen?

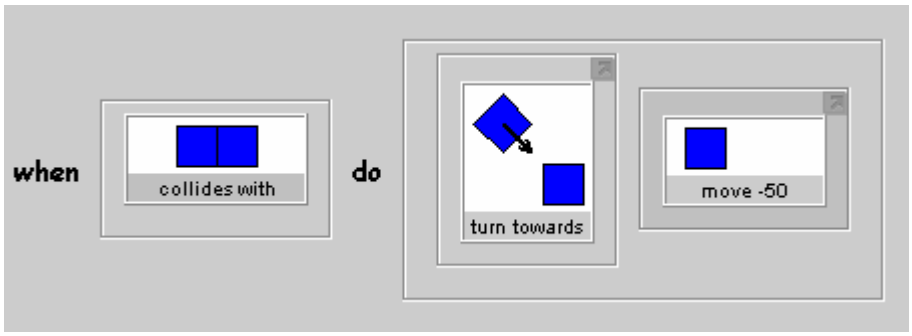


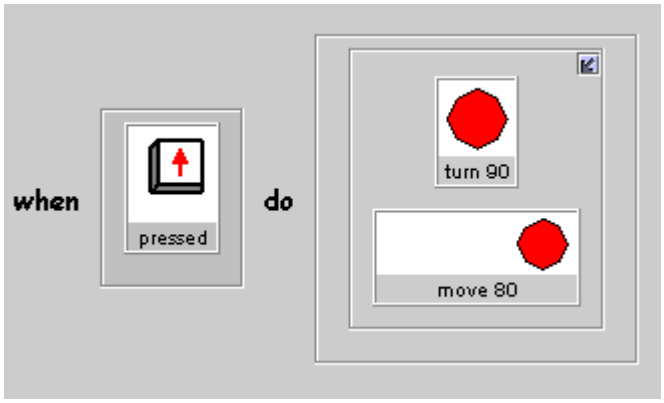
What pattern does the performer make as it moves on the screen?



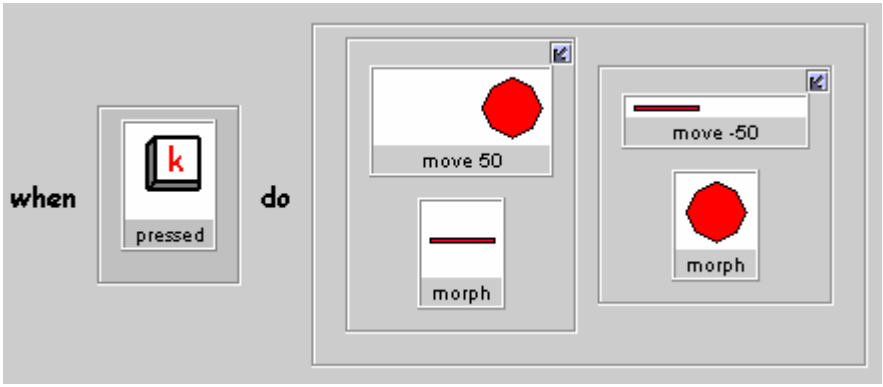






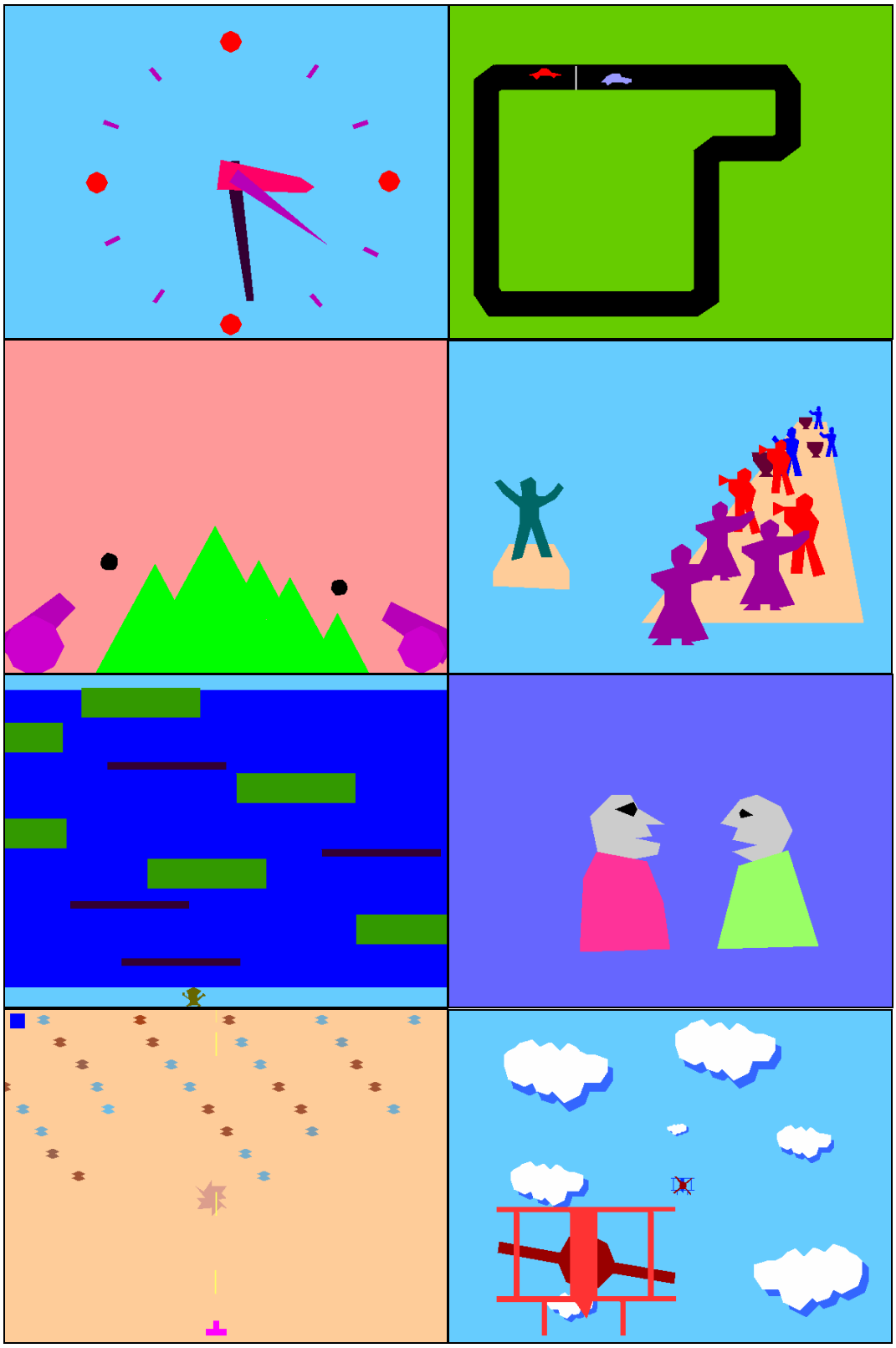


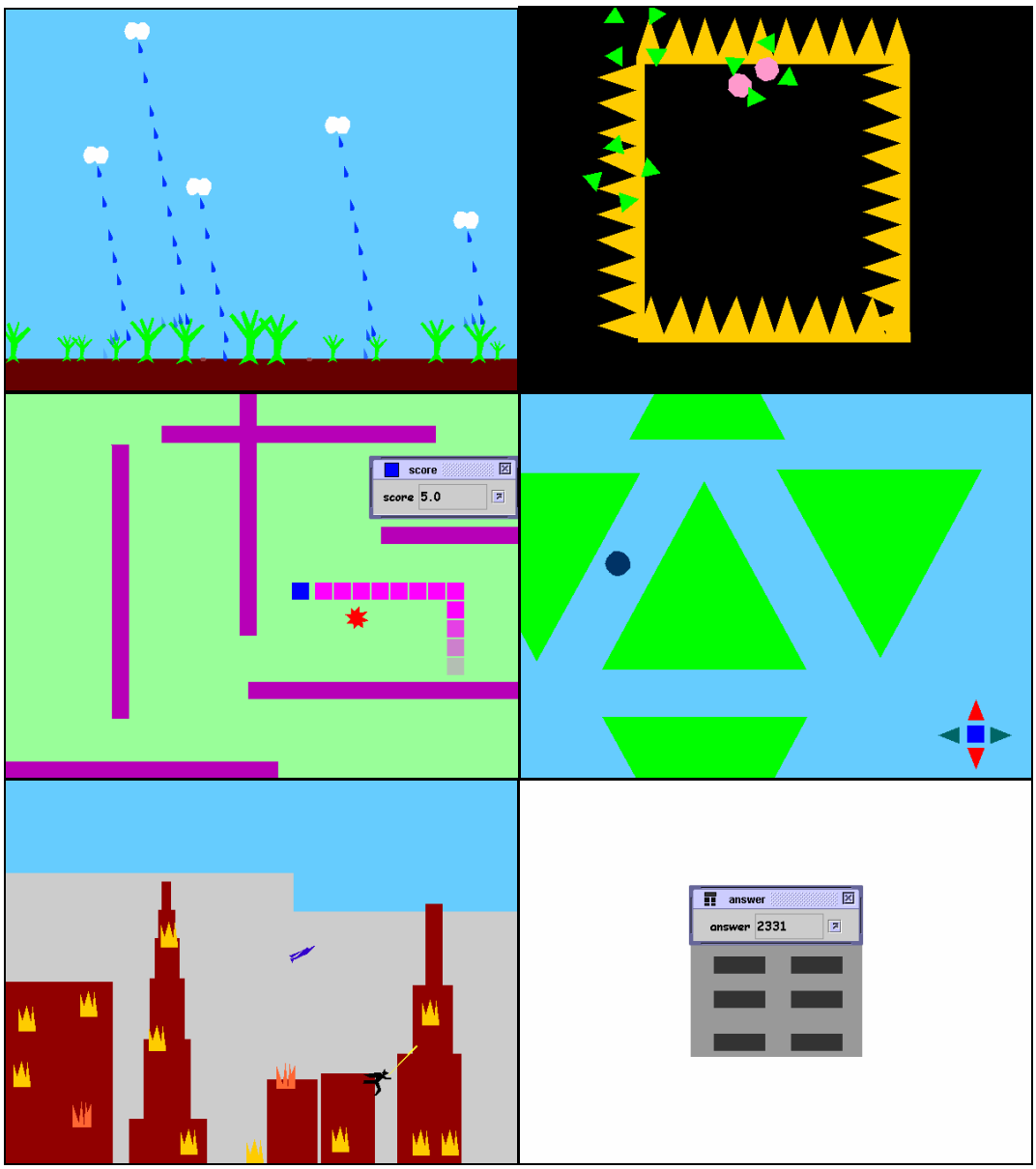
What happens to the performer when the up arrow is pressed three times quickly?


when
k
pressed
do
move 50
morph
move -50
morph

APPENDIX D

Example Icicle worlds, demonstrating the variety of program types that Icicle can represent.




score
score 5.0
answer
answer 2331