

An End-User Programming System
for Constructing
Massively Parallel Simulations

Yoshiki Ohshima

October 2006

Abstract

This dissertation presents Kedama, a programming system for end-users, in particular the children at the school students' age, to enable construction of massively parallel simulations. Kedama allows the user to construct massively parallel simulations in a highly interactive graphical environment. A user-written program can be modified dynamically so that the user can explore the problem domain quickly yet it runs efficiently. In this dissertation, the system, the design goals, the trade offs such as language generality and performance, implementation, and the evaluation of it are discussed.

We aim to satisfy three goals in the user experience of this system as an end-user programming system: 1) decent performance, 2) error-free programming, and 3) immediate feedback. With decent performance, the user can concentrate on building the model and write a straightforward program of the model without committing premature optimizations. With error-free programming, the user can again concentrate on the important part of programming, rather than typing correct keywords, remembering the number of arguments for a command, etc. With immediate feedback, the user can explore the problem domain quickly. Furthermore, it is often the case that the user doesn't have the clear idea up-front how the code should look, and as the user interacts with the model and the program, he gains the understanding of the domain. If the program modification by the user is reflected immediately, the user can try many options quickly. To the author's knowledge, there is no system that satisfies these three goals.

We found that the goals 2 and 3 can be satisfied by taking advantage of an end-user programming system called Squeak eToys. However, for the performance goal, the eToys' original design falls short; eToys cannot keep the interactive response to the user when the number of the user objects reaches a few dozen or more.

Therefore, extending eToys in a way that can handle thousands of objects efficiently without losing its interactive programming nature becomes an interesting challenge. Also, we consider computers that will be used in the

educational scene such as \$100 Laptop Computer and various PDAs. The performance goal should be met even on such relatively slow computers. To achieve the performance goal, we opted to compromise the orthogonality and the generality of the language to a small extent. We will demonstrate this compromise doesn't affect most of examples, or can be worked-around with very small burden.

We experimented with three different parallel execution models called "one-by-one", "simple statement-wise (SSW)" and "predicated statement-wise (PSW)", and settled on the PSW model based on its performance. These models follow a simple abstraction where a command to a group of turtles is executed consecutively by the turtles in the group. In the PSW model, each turtle has a Boolean value called predicate that serves as "masked vector" in High Performance Fortran (HPF) so that the parallel commands within conditional statements can run efficiently.

Based on this fact and Squeak's implementation that allows us to provide functions compiled by an optimizing C compiler, we can provide efficient functions that support the execution of parallel turtle commands. These functions provide efficient vector computation, yet they are used in a way that doesn't interfere with the eToys' dynamic interaction nature.

The performance of the implemented system is reasonably well, and outperforms preceding works, such as StarLogo and NetLogo. In particular, Kedama's performance is linear to the number of turtles; typical examples with a few thousands of turtles run an order of magnitude faster than the preceding works, and gives satisfying performance even on modest hardware. It is also typical that 60% to 80% of the execution time is spent in the C functions; it implies that a similar system written in any language may not be able to give so much drastic performance improvement. On the other hand, restrictions on the generality and expressiveness don't cause practical problems or can be worked-around with small understandable tricks when writing typical examples. With our solution to the performance problem, Kedama becomes a system that meets the stated three goals.

Contents

1	Introduction	1
1.1	Constructivist Theory in Education	2
1.2	Massively Parallel Programming in Education	3
1.3	Desirable Characteristics of an End-User System	5
1.4	Squeak eToys	6
1.5	The Approach	7
1.6	Summary	9
2	Background and Related Work	11
2.1	Requirement Analysis	11
2.2	Related Work	14
2.3	The Overview of Squeak and eToys	18
2.3.1	Squeak: The Language	18
2.3.2	Squeak: The Environment	19
2.3.3	Squeak: The Execution System	19
2.3.4	The Basic of eToys	20
2.3.5	From eToys to Kedama	27
3	The Overview of The Kedama System	29
3.1	Objects in Kedama	29
3.2	Setting Up Objects	32
3.3	Turtle and Breed	33
3.4	Patch Variables	35
3.5	Collision Detection	37
4	Examples in Kedama	39
4.1	Epidemic Simulation	40
4.2	Ideal Gas Tank Simulation	45
4.3	Forest Fire Simulation	51

5	The Static Aspects of Kedama Language	57
5.1	The EToys Language	57
5.1.1	eToys Syntax	57
5.1.2	The Types	60
5.2	The Language Model of eToys	60
5.3	The Addition by Kedama	62
5.3.1	Kedama Syntax Additions	62
5.3.2	The Type Constraints of Kedama Properties	63
5.3.3	The Differences between eToys Objects and Kedama Turtles	64
6	The Parallel Execution Model	67
6.1	The One-By-One Model	68
6.2	The Simple Statement-Wise Model	70
6.3	The Predicated Statement-Wise Model	76
7	Implementation	81
7.1	The Graphical User Interface	81
7.1.1	Kedama World	82
7.1.2	Patch Variable	82
7.1.3	Kedama Turtle Object	83
7.2	The Language Processor	88
7.2.1	Generating the String representation of a Scriptor	88
7.2.2	Creating Parse Tree	89
7.2.3	Attribute Evaluation and Evaluator Generator	89
7.2.4	Parse Tree Transformer	91
7.2.5	CompiledMethod Generation	94
7.3	The Execution Engine	94
8	Evaluation and Discussion	97
8.1	Expressiveness of Language	97
8.1.1	The Language Features	99
8.2	Performance Evaluation	102
8.2.1	Further Performance Improvements	105
9	International Deployment	107
9.1	Multilingualization	107
9.2	Deployment over Various Devices	110

10 Conclusions and Future Directions	111
10.1 Conclusions	111
10.2 Future Directions	112
10.2.1 Explore Different Languages	112
10.2.2 Transition Between “Big” and “Small” Objects	113
10.2.3 More Types for Patch Variables	114
10.2.4 Multiple Worlds Work Together	114
10.2.5 3D	115
10.2.6 Fields	115
10.2.7 Hardware Dependent Optimization	116
A Squeak	117
A.1 Squeak: Its Origin and Concept	117
A.2 Squeak Language	118
A.2.1 Squeak Syntax	118
A.2.2 Messaging	119
A.2.3 Classes in Squeak	119
A.2.4 Everything is an Object	121
A.2.5 Dynamic Modifications	121
A.2.6 Context Aware Exceptions	122
A.2.7 Collections	123
A.3 Squeak Environment	123
A.3.1 Computer as an Object Environment	123
A.3.2 Development Tools	124
A.3.3 GUI Frameworks	125
A.4 Execution Model and Implementation	127
A.4.1 Execution Model	127
A.4.2 Squeak Virtual Machine	127
A.4.3 Squeak VM Instruction Set	128
A.4.4 Squeak Memory Management	128
A.4.5 Accessing the Platform Functions	129
A.4.6 VM generation and Performance Primitives	130
A.5 Squeak Virtual Image and Object Format	131
B The List of Features of Kedama Objects	133
B.1 The Kedama World Object	133
B.2 Patch Variables	135
B.3 The Turtles	137
C The Syntax Transformation Rules	139

D	The Execution Engine for Kedama	145
D.1	Numeric Operations	145
D.2	Logical Operations	147
D.3	Predicated Array Assignment	148
D.4	Primitives for Turtle Actions	148
D.5	Other Primitives	150

List of Figures

2.1	A screenshot of eToys session.	22
2.2	An object with halos.	23
2.3	A script that makes the Car object go circle.	24
2.4	A script to drive the Car object with Wheel object's heading.	26
3.1	The schematic view of the system.	30
3.2	The graphical representations of a Kedama World (left), two patch variables (middle), and two turtle exemplars (right).	31
3.3	Instantiating a Kedama World from the standard Object Catalog tool.	32
3.4	The viewer for the Kedama World object.	32
3.5	A Kedama World, a patch variable, and an exemplar of a breed.	33
3.6	A Kedama World with ten turtles of a breed.	33
3.7	A script with <code>forward by</code> and <code>turn by</code> command.	34
3.8	The execution flow in <code>script1</code> . (See Chapter 5 for detail.)	34
3.9	The tile implementation of <code>script2</code>	36
3.10	A fun Kedama project that puts meteor-like tails to the turtles.	36
3.11	The execution flow in <code>script2</code>	37
3.12	The tile implementation of <code>script3</code>	37
3.13	The execution flow in <code>script3</code>	38
4.1	The infected property of Boolean type in the exemplar's viewer.	40
4.2	The <code>setup</code> script for epidemic simulation.	41
4.3	The <code>oneStep</code> script of the epidemic simulation.	42
4.4	The initial state of the epidemic simulation.	42
4.5	The progression of the simulation.	43
4.6	The <code>plotter</code> 's script and the resulting graph.	43
4.7	The graphs for 200 villagers and 1,000 villagers.	44
4.8	The graphs for 200 villagers with weak contagiousness and strong contagiousness.	45

4.9	An illustration of the narrow perspective. Through this time window, it is hard to tell what is going on.	45
4.10	Dropping a Sketch onto a Kedama World creates turtles lined up with the bitmap.	46
4.11	Create the ceiling of the simulated gas tank from a sketch. . .	47
4.12	The <code>setup</code> script for the gas tank simulation.	48
4.13	The <code>oneStep</code> script for the gas tank simulation.	48
4.14	The equilibrium with 2,000 molecules.	49
4.15	The equilibrium with 1,000 molecules.	50
4.16	The equilibrium with 500 molecules.	50
4.17	Fill a Kedama World with turtles by dropping a big sketch. .	51
4.18	<code>savePosition</code> and <code>restorePosition</code> for a cellular automata type simulation.	52
4.19	The <code>calcFlamelevel</code> script for the forest fire simulation.	53
4.20	The <code>oneStep</code> script for the forest fire simulation.	53
4.21	A forest initialized with 63% of tree.	54
4.22	A burning forest.	54
5.1	The EBNF of the eToys tile scripting language.	58
5.2	The EBNF of the addition of Kedama.	63
6.1	The execution thread in <code>script1</code>	68
6.2	The execution thread of <code>script1</code> in SSW.	71
6.3	The execution thread of <code>script4</code> in SSW.	72
6.4	The execution thread of <code>script5</code> in SSW.	73
6.5	The execution thread of <code>script7a</code> in SSW.	74
6.6	The execution thread of <code>script8</code> when called from <code>script7a</code> in SSW.	75
6.7	The execution thread of <code>script8</code> called when called from <code>script7b</code> in SSW.	76
7.1	The organization of classes for a breed of turtle.	84
7.2	The straightforward representation of array of turtles.	85
7.3	The horizontal representation of array of turtles.	85
9.1	The original hierarchy around <code>String</code>	109
9.2	The current hierarchy around <code>String</code> (indentation denotes the superclass-subclass relationship).	109
A.1	An inspector on the <code>Display</code> object.	124

C.1	The transformation rules for statements.	141
C.2	The transformation rules for variables.	142
C.3	The transformation rules for blocks.	142
C.4	the transformation rules for message sends.	143

Acknowledgment

I would like to express my gratitude here. First of all, I would like to thank Professor Masataka Sassa for taking the responsibility and providing the support as the advisor of my dissertation. I would also thank my dissertation referees, Professor Etsuya Shibayama, Professor Satoshi Matsuoka, Assistant Professor Shigeru Chiba and Assistant Professor Ken Wakita for their helpful comments and suggestions.

This work, both writing software and even more notably writing this seemingly redundant document in a foreign language, wasn't conceivable without help and encouragement from my colleagues all over the world; among others, Alan Kay, Kim Rose, John Maloney, Dan Ingalls, Ted Kaehler, Scott Wallace, Michael Rueger, Andreas Raab, Ian Piumarta, Takashi Yamamiya, and Kazuhiro Abe. I would like to “pay it back” to them in the coming years.

Of course, many other fellow students, fellow interns, friends, colleagues at the Sassa-Wakita Laboratory of Tokyo Institute of Technology, Walt Disney Imagineering R&D, Twin Sun, Inc., Viewpoints Research Institute, The Apple Hill Learning Lab attendees, and the Squeak community has given me great support. Their feedback and suggestions gave me many good ideas.

The discussion with the StarLogo team members, Mitch Resnick, Brian Silverman, and Andrew Begel, have been invaluable. I would like to thank them as well.

My family, late father, mother, and two sisters must have been helping me in many ways which I often didn't notice. (どうもどうも.) Also, my little nieces and a nephew, who can't possibly care less what I do for work but who think I'm their buddy (and I am), have been my “motivating officers”.

Finally, I would like to thank my wife, Haruko, for waiting such a long time. (お待たせしました！)

Chapter 1

Introduction

This dissertation presents an interactive programming system for end-users, typically the learners at the age of students at various levels of schools, to write simulations with massively parallel objects. In this system, named Kedama, the user can create and manipulate thousands of parallel objects in an interactive programming environment.

Kedama aims to provide a *real* usable system that is based on good educational theory. Such theory and its relationship with Kedama are explained in Section 1.1.

The idea of massively parallel objects is a powerful tool to model a wide variety of phenomenon in physics, mathematics, and the social sciences. The idea of massively parallel objects and its application as an educational tool is explained in Section 1.2.

We will identify three goals that an end-user programming system for parallel programming should satisfy in 1.3. These are 1) decent performance, 2) error-free programming, and 3) immediate feedback from the system. Among these three issues, the last two can be addressed by using an existing system called Squeak eToys. Therefore, addressing the performance issue while keeping the good aspects of eToys is the main goal of this work. Needless to say, providing good performance for a massively parallel system, where many thousands or millions of objects need to be controlled, is an interesting challenge.

The work of Kedama is based on Squeak and Squeak eToys [1]. Squeak eToys is described briefly in Section 1.4 and in Chapter 2.3.

The work of Kedama, as a computer science project, can be considered as a kind of parallel and visual programming language.

After providing such background, the approach of this work is briefly

described in Section 1.5. The parallel execution model we use is named the “predicated statement-wise model”, whose execution order is analogous to the SIMD abstraction but has some differences in particular the way that the visibility of intermediate data is handled.

Finally, the summary of this work is given in Section 1.6.

1.1 Constructivist Theory in Education

Kedama tries to draw upon many of the good ideas from the long history of educational theories. In the following, the “constructivist theory”, which Kedama is mostly based on, is briefly discussed to set the background of this dissertation.

While it would be ideal if we would be able to pick “the best” educational theory, evaluating an educational theory or method is not simple. There have been many different theories on education proposed, but there seems no “perfect” one that everybody can settle. The problem is that each of such theory always works on somebody; since all learners have different interest and different ways of thinking. Any theory is suitable for some learners. When one educator claims that his educational method is better than others, supporting the claim objectively is difficult.

The above-mentioned difficulty arises from two facts. One is that education has many different purposes. Another is that individual students have different interests and different ways of thinking, therefore single standard teaching material would not appeal to most students. Although the first fact is somewhat offset by the relatively limited scope of Kedama, which is on science, mathematics and related fields in school students, the second fact still remains.

One way to mitigate the difference among students is to allow individual students to work on different material that matches the student’s particular interest, yet to keep the student on the line of the subject’s goal. Furthermore, if a learner is allowed to construct by himself the curriculum and knowledge during the learning process (with help and facilitation from teachers), the student can keep his interest on the subject, gain deeper understanding of it, and go beyond the given information. This idea, that each student constructs the material by himself while learning, is called the “constructivist learning theory”.

The idea can be traced back to Maria Montessori in late 19th century and early 20th century [2]. From careful observation, Montessori built her theory on early childhood education. The observation was that the child

has intuitive aim to develop himself. In other words, children are set up by nature to learn the surrounding environment as something to cope with through intimate interaction. Montessori's theory was, if children were put in the right kind of educational environment, they would interact with that environment just as they do with ordinary environment and learn the interesting ideas behind the environment. Montessori built a rich set of carefully designed "toys" that have hidden educational merit, and let children choose (but carefully) the toy they would like to play with. With close guidance from an adult, the children can sustain the joy of learning.

More formalization of the idea is attributed to Jean Piaget [3]. Based on his cognitive psychology research with children (including his own), he developed a theory on the cognitive development of children.

Jerome Bruner [4] also played an important role to further develop the line of thought. A major contribution is to formalize the importance of sequencing the course of learning. A famous quote: "We begin with the hypothesis that any subject can be taught effectively in some intellectually honest form to any child at any stage of development." summarizes the capability of children and suggests the idea of "spiral curriculum", where the same topic is visited from different angles and depth to lead the learner to grasp the full formal apparatus that goes with them.

Constructivism, the approach that is based on the constructivist theory, has been used in actual schools, and this philosophy has been influencing many educators around the world [5]. Seymour Papert coined a similar term "constructionist approach", which implies a more active role of learners as a designer and constructor of an entity or ideas that can be shared with others. Based on this idea, Papert and Feurzeig created the Logo system in '67 [6], in which young learners can construct a program that is specialized for creating geometric shapes. Logo users can share their program and the results from the program as their project. That means, they learn not only in the programming environment, but also with the environment.

Kedama draws upon the idea of constructivism and tries to provide the "environment" in which the student can explore with dynamic simulations and gain the knowledge from them.

1.2 Massively Parallel Programming in Education

Many physical, biological and social phenomena, as well as mathematical concepts can be modeled with a massive number of objects, simple rules that the objects follow, and the interaction between them. Often, there is

no central or common knowledge shared by the objects in such system (i.e. “decentralized”). However, the variation in the elements’ initial state and interaction between them produce dynamic phenomena that are often unpredictable from the rules. Such unpredictability is also known as “emergence”, and the systems that give it are often called “complex systems”.

One reason why emergent behavior occurs is that the number of interactions between elements of a system increases combinatorially with the number of elements; some micro-level effects on one element may cause cascading effects on others and the collective effects can be seen as completely new behavior from the simple rules. For example, suppose we have a plastic bottle, in which some kind of gas molecules are contained. At the micro-level, each molecule is moving in its own direction as a solid object and bumping each other and the inside of the container. However, at the macro-level, an individual molecule’s movement itself is irrelevant, but the massive number of molecules hitting the inside of container can be measured as “pressure”, which is a very different phenomena.

One of the earliest formalizations of emergent behavior in computer science field was the cellular automata of Codd in ’68 [7] although the idea of unexpected patterns emerging from simple rules was found in the bit-registers and automata theory. This work led to Conway’s famous “Game of Life” [8].

An early article in physics that discusses the complex system was written by Nobel Prize winner Philip W. Anderson. The article in ’72 was (rightly) titled “More Is Different” [9]. In this article, he identifies two trends in science he calls “intensive” and “extensive” research. According to him, “intensive research goes for the fundamental laws, and extensive research goes for the explanation of phenomena in terms of known fundamental laws. One view held among some researchers is that the intensive research can explain all phenomena in the world. However, he elaborated the difference of reductionism and constructionism. In short; “the ability to reduce everything to simple fundamental laws does not imply the ability to start from these laws and reconstruct the universe.” He uses a few examples in chemistry and how new behaviors emerge at a bigger particle level from the simpler particle level. By the way, I think this note on reductionism can be seen as an admonition to the computer science field.

With help from the advancement of chaos and fractal theories in the late 80’s and 90’s, the complex system models are used for many real world phenomena. The idea to produce complex results from some simple rules has been applied in many areas such as photo-realistic computer graphics and physics simulation.

Massively parallel simulations and emergent behavior from a simple program is also attractive to educators who try to use new tools in classrooms. If students could construct their own simulations and explore the problem domain, they would reach a better understanding of the simulation and simulated phenomena. A computer is an ideal tool to simulate such systems, because the combination of relatively simple rules (program) and a large amount of data is a good match with a computer.

From this observation, and the powerful idea of constructivist theory, the author is interested in the idea of implementing an interactive massively parallel programming system for school students.

1.3 Desirable Characteristics of an End-User System

Upon designing a programming system for end-users, the author claims that such a system should satisfy the following three characteristics.

Decent Performance It is desirable to let the user concentrate on making the model and write a straightforward program from the model. However, it is often the case that *premature optimization* is required to gain enough performance. Premature optimization is bad because it can obfuscate the relationship with model and code, and requires knowledge that is irrelevant from the actual content.

Error-Free Programming It is desirable that the user doesn't suffer from the nuisance of the programming language and environment. The user shouldn't have to type in a lot of code correctly, or is required to deal with the data type, number of arguments to a function. If the programming system offers good support for the user, the user can concentrate on making the model and write a straightforward code from it.

Immediate Feedback The learner, or the user, often doesn't have the prior knowledge of the final shape of the program while constructing a model. Also, a typical simulation often contains some artificial parameters that should be tweaked by running a program with different values. If the user can change the program and parameters in it easily and the system responds quickly and gives *immediate feedback* to the user, the user can try different programs quickly. It also allows the user to try "what-if" simulations. If the user can change the code in

the middle of a simulation while the state of the ongoing simulation is kept, the user can often learn the different program's behavior with some interesting initial conditions. This can help the user to learn the model even more deeply.

When the author surveyed the educational programming environment with massively parallel programming capability, there wasn't a system that satisfied these three characteristics. With this observation, the author decided to implement a programming system that satisfies these criteria. More detailed survey and discussion is described in Chapter 2.

1.4 Squeak eToys

As a platform for implementing such a system, the author has chosen a system called Squeak eToys ("eToys", hereafter) [10][11]. EToys is a visual, tile-based scripting system designed primarily for fifth and sixth grade (11 and 12 years old) children. In eToys, the user can directly manipulate graphical objects on screen, and construct a program by using symbolic and graphical representations of objects.

In eToys, the user drags and drops fragments of a program represented as graphical "tiles" to construct a program. Examples of such tiles are a command to an object, a variable reference, literals including numbers, a template for conditional statements, etc. When the user tries to drop a tile onto a "drop-zone" in another tile, type checking is performed and the tile being dropped is accepted only if the types agree. With this type checking, the code is ensured to be "correct" and not to raise type-related runtime errors. (There may be runtime errors such as zero division, but such errors are handled with a user-friendly dialog.)

More notably, the change to a program by the user is reflected immediately in eToys. The user can change a running program (imagine a program that is repeating a loop), and the program changes its behavior with the current state.

EToys is heavily influenced by constructivist theory. As its name suggests, eToys serves as a kind of toy with which children can play and have fun, just as Montessori's toys do. However, it is designed to let children build simulations and programs so that they can engage the mathematical and scientific studies.

Logo also influenced the design of eToys. Logo's primary idea was to be able to teach "powerful ideas" to children. One of such ideas was that the user writes a script to describe a geometric equation in differential form

(often only with first order expressions, or additions), and the computer does the integration by executing the script repetitively over time. EToys inherits this differential geometry idea from Logo and adds more flexibility, rich graphics and sound. EToys also encourages exploration by the user more than its predecessors. The user can dynamically manipulate the properties of objects and their behavior at runtime, and the objects follow the change immediately.

Since its public release in '99, eToys has grown in popularity and is used officially by schools around the world; The US, Spain [12], Japan [13], Germany, Canada, South Korea, and many more countries. Also, there are several books published about eToys in different languages [14][15].

Kedama is implemented as an extension of eToys to leverage the familiarity with the existing eToys users. Kedama is incorporated into the official Squeak distributions, so the world wide users have access to it.

In chapter A, a typical user experience of eToys is explained.

1.5 The Approach

The goal of this project is to provide a system that satisfies *all* three desirable characteristics identified in Section 1.3, which none of the existing systems do.

As described in Section 1.4, eToys can satisfy two of the three characteristics; “Error-Free Programming” and “Immediate Feedback”. Therefore, by adding reasonably fast massively parallel capability to eToys in a way that doesn’t sacrifice the favorable characteristics, we would be able to provide a system that satisfies all three characteristics to the users.

EToys is designed to control user objects that have a “big” graphical appearance (i.e., tens to hundreds of pixels in their dimensions). Each time an object is rotated, for example, the graphical bitmap that represents the object is rotated, and the certain area of the screen is invalidated. Also, the computation of the coordinate calculation is done with boxed representation of floating point numbers. As a consequence, once the number of the user-created objects exceeds a few dozens, eToys becomes too sluggish to provide immediate feedback. This is clearly a problem to be solved.

From this observation, we decided to extend the eToys and add the capability to write massively parallel programs. The extension should provide decent performance, but at the same time, it should be implemented so that the good characteristics of eToys are retained; i.e, the dynamic interaction including code modification should be possible and the same tile-scripting

system of eToys should be used.

If we make a system that has “decent performance”, how decent should it be? Even on a decently fast personal computer, handling a few thousand objects and showing smooth animation with them (that is important for the immediate feedback goal) cannot be achieved if the execution of the program is relatively slow. If we consider the slower platforms such as \$100 laptops, it is more true. As shown later, it is often the case that an example of a simulation needs 10,000 turtles. If the user would like to perform 20 frames per second animation from the simulation and the computer has a 1.0 GHz Intel processor, the CPU cycles that a turtle can use in a frame is:

$$(1,000 \times 1,000 \times 1,000)/20/10000 = 5,000(\text{CPU cycles/frame/turtle})$$

Considering the fact that a call of `sin()` and `cos()` can take up to 112 CPU cycles on recent Intel x86 chips, 5,000 CPU cycles for all actions by turtles as well as sending the resulting pixels to the display is rather limiting. Another way to look at this number is that the program written by the user (in the tile-scripting system) should run about at the speed that is comparable with the naively written program in the C programming language. This sets the criteria of “decent performance”.

To achieve this performance goal, we opted to compromise the orthogonality and the generality of the language to a small extent. As shown in Chapter 4, most of the examples are not affected by such annoyance or can be worked-around with very small burden.

We looked at different parallel execution models and their trade-offs. The ancestors of this work, StarLogo and NetLogo use a parallel execution model that can be called a *multi-threaded model*. In this model, a thread is allocated to each turtle upon the invocation of a script, and these threads execute the content of the script concurrently. This design focuses on giving the user an illusion that each turtle is an autonomous and independent creature at the cost of a performance penalty. We would like to keep this illusion, but we think that it can be done in a different way. Because the screen update is synchronized with the script invocation in Squeak, the execution order of the effects in a script invocation is not visible to the user. This fact gives us the freedom to opt for an alternative execution model to reorder the command executions in a script for parallel turtles to gain better performance.

We experimented with a few different execution models. Three notable ones are called “one-by-one”, “Simple statement-wise (SSW)”, and “predicated statement-wise (PSW)”, and settled on the PSW model. The PSW execution model is analogous to the SIMD abstraction with a Boolean value

called *predicate* attached to each turtle. The predicate serves as the “masked vector” in High Performance Fortran (HPF) so that the parallel commands embedded in the conditional statements can run efficiently. Note that the term “SIMD” is used only analogously; while the term “SIMD”, in a strict sense, implies that the data consistency (i.e., the intermediate state from one turtle is not visible to other concurrently running turtles), it simply is used to explain the execution order where one thread executes the actions from a statement is completed before advancing to the next statement. In the other words, the effects from a turtle in a statement may be visible to subsequent turtles in the same statement in Kedama. The detail of execution models is discussed in Chapter 6.

Another restriction on Kedama turtles is their generality; from the user’s point of view, a turtle in Kedama is slightly less capable than other eToys’ objects. The restriction is explained in Chapter 5 and 6.

The slightly relaxed execution model and the restriction on the turtles’ generality let us implement a simple and efficient execution engine. We have extended the Squeak virtual machine (VM) and provide C-compiled *primitives*. A primitive for Kedama typically iterates over homogeneous turtles to perform a command on them. The representation of the turtles heavily utilize the *homogeneous arrays* that can store an array of data in a C-compatible bit pattern. With this use of homogeneous arrays and primitives, most of the heavy computation is done by C-compiled functions.

In a typical example of Kedama, 60% to 80% of execution time is spent on such primitives. This implies that it may be hard to implement a similar system that runs twice as fast as Kedama in any language. Quite possibly, the overhead of the dynamic interactive objects implemented in some other languages would make the performance of such a system comparable to Kedama’s.

1.6 Summary

The contribution of this work is summarized as follows.

We identified three characteristics that an end-user and educational programming system should satisfy in Section 1.3; these are 1) decent performance, 2) error-free programming, and 3) immediate feedback to the user interaction. However, none of the existing systems meet these three characteristics at the same time.

Two of the three characteristics, “error-free programming” and “immediate feedback”, were satisfied by taking advantage of the eToys system. We

then added a massively parallel programming capability to eToys.

A parallel execution model was designed and implemented. Unlike the models used in similar systems, Kedama adapted a model analogous to the SIMD abstraction with some relaxation. Also, there are some restrictions put on Kedama turtles compared to other eToys objects. The implementation provides decent performance that is comparable to a system that would be written (naively) in the C language. Most notably, the implementation is scalable in terms of the number of parallel turtles; unlike StarLogo or NetLogo, its performance is linear to the number of turtles. Given an example with more than a few thousand turtles, Kedama runs about one order of magnitude faster than NetLogo.

The massively parallel capability was added in a way so that it doesn't sacrifice the existing favorable characteristics of eToys. Tile-based error-free programming, dynamic modification of a program and immediate feedback for the modification are preserved. In fact, the language is kept unchanged from eToys so that existing users of eToys can experience a smooth learning curve.

Incidentally, the language of eToys is more suitable than Logo to extend to a massively parallel version. In the "object-oriented" syntax in eToys, the "receiver" is always specified for all commands. It makes clear to the user who performs the command.

Furthermore, eToys runs on top of an open-source highly portable virtual machine. For technical and political reasons, not many non-open source languages will run on future large quantity platforms such as the \$100 laptop computer and PDAs. Kedama's primitives are written so that it doesn't hamper the porting effort, yet it is quite possible to substitute primitives with the functions with hardware-dependent optimization because such primitives typically contain small inner loops with uniform data arrays.

With the consideration of a parallel execution model and implementation of this work, we successfully demonstrated that a programming system for a massively parallel simulation system can be built on a tile scripting system and the user program written in it can run very efficiently. As the supporting evidence of its usefulness, the system is actually deployed to the world as a part of eToys programming system, and used in many schools.

Chapter 2

Background and Related Work

In this chapter, the background and requirement analysis, related work, and a system we choose to use as the basis are described. In Section 2.1, the requirements of a special kind of massively parallel programming system whose target audience is school age students are discussed. Also, the related work are analysed based on the requirements in Section 2.2. We chose to use the “Squeak” system and “Squeak eToys” system as the basis of Kedama. In Section 2.3.1, Squeak and Squeak eToys are described.

2.1 Requirement Analysis

Our system aims rather specific audience. The audience is non-technical learners at school ages, and a typical program in the system is such that manipulates thousands to millions of objects. This specific focus brings up some interesting requirements. As described briefly in 1.3, there are characteristics that should be satisfied by a system for this particular domain:

Decent Performance It is desirable to let the user concentrate on making the model and write a straightforward program from the model. However, it is often the case that *premature optimization* is required to gain enough performance. Premature optimization is bad because it can obfuscate the relationship with model and code, and requires knowledge that is irrelevant from the actual content.

Error-Free Programming It is desirable that the user doesn’t suffer from the nuisance of the programming language and environment. The user

shouldn't have to type in a lot of code correctly, or is required to deal with the data type, number of arguments to a function. If the programming system offers good support for the user, the user can concentrate on making the model and write a straightforward code from it.

Immediate Feedback The learner often doesn't have the prior knowledge of the final shape of the program while constructing a model. Also, a typical simulation often contains some artificial parameters that should be tweaked by running a program with different values. If the user can change the program and parameters in it easily and the system responds quickly and gives *immediate feedback* to the user, the user can try different programs quickly. It also allows the user to try "what-if" simulations. If the user can change the code in the middle of a simulation while the state of the ongoing simulation is kept, the user can often learn the different program's behavior with some interesting initial conditions. This can help the user to learn the model even more deeply.

Also, there are some additional criteria for making a practical system that can be deployed to the real educational environments; most important ones among these are portability and expressiveness and understandability of the language.

Portability The system should run as wide variety of platforms as possible, as the computers in classrooms over the world are often different from mainstream computer platforms like Windows. At the same time, we cannot assume to have super computers for each student. A typical computer in the classrooms is modest, cheap one.

Good Enough Expressiveness and Understandability The system is expected to provide good expressiveness that covers typical examples the user would write. At the same time, because the system deals with end-users, the semantics of the language should be simple and *understandable* yet it should not prevent providing good performance; the system should aim to be more understandable and familiar to the end-users, rather than having vast amount of features. We would sacrifice the highly tuned optimization for understandability of the language.

As you may have noticed, some of these goals conflict with each other. The combination of good performance and immediate feedback is an example; a programming environment that tries to give immediate feedback may

not have enough time to optimize. So are dynamic modification of running code and a understandable language vs. performance.

For gaining maximum performance, one would consider using a parallel computer. In fact, StarLogo, the first attempt to provide an end-user accessible programming system for massively parallel simulations, was implemented on the Connection Machine. However, we cannot assume to make a super computer available for each learner. Therefore, our requirement on performance is that the system should run on commodity computers (i.e., single processor computers), but it should still be reasonably optimized for such platforms.

For providing error-free programming, one can consider a language with some kind of type system or some editing support from the development environment. However, as long as the user has to use a keyboard to write a program, the type system or the support from the development environment cannot provide maximum benefit. Rather, we look at the visual programming language concept; in a visual language, the user constructs a program by combining graphically represented program fragments (often called tiles or blocks) with a pointing device. In such an environment, type checking could be done when the user tries to combine two tiles. More notably, the user doesn't have to remember or type in right keywords and variable names.

The immediate feedback goal has a few different aspects. One is that the state of running programming is visible to the user real-time so that the user can immediately see the state of simulation he writes. Another is that when the user's interaction takes effect immediately. The need for (efficient) visualizing a running program favors a good integration between the parallel language and the graphics system. To provide good dynamic interaction, the system should not let the user go through the "edit-compile-test" cycle.

From the observations above, refined requirements for the system to be implemented can be summarized as follows:

- Reasonable performance on single processor computer.
- Visual programming in an end-user accessible programming environment.
- Real-time graphical rendering of the simulation states.
- Smaller language feature set for understandability.
- No edit-compile-test cycle.

The system this dissertation describes, Kedama, tries to meet all of these requirements. In the next section, we visit existing languages and systems to see how they match our requirements.

2.2 Related Work

In the 90's the trend of parallel programming made an interesting twist in a different area when Resnick proposed an educational system called StarLogo [16]. StarLogo and NetLogo [17] are designed for high school students' use, and these systems have been accepted by many educators and students. The user writes Logo-like program code to control hundreds to thousands of objects (called "turtles"). The execution state of the program is animated on screen without any additional efforts by the user. StarLogo was first implemented on the Connection Machine 1 [18] super computer, but of course Moore's Law made it possible to later run such systems on personal computers [19]. NetLogo can be considered as an improved system for StarLogo in terms of UI and performance. StarLogo and NetLogo have been successfully used in schools, but these systems don't provide full immediate feedback as the user has to write program in text with keyboard and go through the edit-compile-test cycle, and the performance is not satisfying (more details are described in Chapter 8). Also, these systems have separate panes for writing textual code and running the simulation. The user has to switch back and forth between the panes, and when the user does so, the program stops running. Nonetheless, these systems were the source of inspiration of Kedama.

Another problem of StarLogo and NetLogo is the language. The syntax of Logo, on which StarLogo and NetLogo are based, is adequate when there is only one entity in the system with which the user interacts. For example, if the user writes a `setColor` command in Logo, there is no ambiguity with respect to the object that will change color. Once the system starts having other kinds of objects, the "receiver", or the object that performs the specified command, has to be specified. However in the systems that rely on the Logo syntax, the decision is made implicitly.

Here let us visit other educational programming languages.

ToonTalk [20] is an educational programming environment. Its language is declarative and logic-based, but there is a cartoon-like or toy-like interface that shares the same spirit of eToys and Kedama. Because the language is logic-based, the parallel execution semantics could be very clean. While ToonTalk doesn't provide "massively" parallel execution, one could imagine

a logic-based parallel language like KL1 and its portable implementation KLIC [21] with an accessible interface would be interesting for end-users' use.

For educational purposes, traditionally languages like Pascal were used and later other languages like Java become popular in the area. However, even for university students, some concepts such as loop and iteration can be obstacles of learning. So is the magic words such as “`public static void`”, etc. There are some attempts to “wrap” such idiosyncrasies of languages with some accessible environments. For example, Nigari [22] is an example. Nigari has graphical user interface, and each object generally has a graphical look, and has basic properties defined. The user only write the interesting part of the program in the environment and the environment generates the Java code. By assigning a thread for each user-object, Nigari alleviates the multi-thread programming burden from the user. On the other hand, the multi-thread model reaches its limit when the number of objects is about a few thousands. To handle more objects, better mechanism is needed.

Scheme is another popular language in the university education. It has clean semantics and, more notably, its meta-language features help the learners to learn various concepts of programming languages. If one applies the constructivism theory to the programming language education, the best way to learn the programming language concepts is to make them. Also, there are again integrated environments such as DrScheme [23] that try to make the learning curve smooth. While these systems are successful in their domain, they don't provide the massively parallel programming or visual rendering of running simulation easily.

There is a genre of visual programming environments, where the user specifies the transition rules between “before” and “after” states graphically, and the system looks for the applicable rules to current state from the set of rules and applies them. Along this idea, Tableau [24] was the early experiment by the creators of Smalltalk, and it later became KidSim [25] and StageCast [26]. Viscuit [27] introduces a fuzzy matching on the rules so that the user can provide flexible rules. These systems lower the first barrier of programming greatly, but they reach their limit rather early; as long as the model that a program describes matches simple state transitions, these systems work great. However, once the model requires some abstraction, even as simple as numbers and arithmetic, the state transition rules, especially with fuzzy matching, cannot describe the model. One could imagine extending these systems to accommodate massive number of objects, but the language need to support some abstract concepts for writing meaningful simulations.

SOARS [28] is an agent-based modeling system with an end-user accessible visual interface called VisualShell [29]. The most notable aspect of SOARS is that its model doesn't rely on the grid cells but has a concept of "spots". While VisualShell provides graphical editing of some part of a model, but the primary specification of a model is in text. VisualShell is more of an organizer of text descriptions, and the modification of a description requires a few steps to be reflected. SOARS has been used at some university classes, but for younger audience, its complexity would be overwhelming.

Alice [30] is an educational environment that is designed to learn programming with 3D objects. In the later version of Alice [31], a tile-scripting system is incorporated. A notable design trade-offs is that the system has tiles called `doTogether` and `doInOrder` to specify different parallel executions. If the user puts two command enclosed by a `doTogether` block, these two commands take effect concurrently. As described later chapters, Kedama opted not to introduce such construct to keep simplicity. On the other hand, there are cases when such explicit concurrency description is helpful to specify different concurrent execution. Another difference is that Alice doesn't handle so many objects.

AgentSheets [32] shares a lot of characteristics with our system. It has tile-based programming system and allows dynamic modification of program. It is based on the 2D grid model as our system. However, the target is not "massively parallel"; the size of a grid is rather large, and overall the system is designed for creating a model with a few hundred cells. Also, unfortunately, it is a commercial product.

Let us look at the related work from different angle. The work of this dissertation can be seen as a new kind of massively parallel programming system. However, the main focus of the work is the usability and expressiveness in an educational situation, rather than competing with the state of art high performance computing. Nonetheless, it would be worth putting the work in the historical perspective of parallel computing and end-user oriented computing.

From the early days, one of the computer's major applications has always been high performance computing with massively parallel data. The nature of computing and the human being, where relatively simple rules are provided by programmers for relatively uniform, massive amount of data, is suitable to support it. (It is said in [33] that it dates back to ILLIAC IV [34] and the early Parallel Fortran (IVTRAN) [35].)

Moore's Law gave us the exponential growth of the amount of data that a computer can hold. Since the ability of a human being can't catch

up with that rate, managing continuously growing complexity requires the automation of the programmers' work.

There is a history of parallel programming language research [36, 33, 37, 38, 39, 40]. Naturally, many of these languages are designed to write software for parallel computers, and aim to provide optimized performance. There are often concurrency control features are built-in the languages and the programmer has to use these control structures properly. These languages are suitable for the audience from undergraduate students in computer science and above, but certainly not for the non-technical high school students.

Some parallel constructions in past languages provide simple and effective "default" parallelism, such as the concept of Concurrent Aggregates (CA) [37]. An aggregate in CA provides homogeneous collection of objects, and provides a kind of facade of concurrency. The system of this dissertation has simpler version of aggregates, through its inspiration from StarLogo, where there is only one-level of aggregated collection allowed only one type of parallelism is created by the aggregate.

As for an implementation technique for efficient execution, our system borrows well-known techniques from the past systems such as HPF [33]. As described later, our system only has one control structure in the language; a conditional statement. By using the similar idea to the masked-vector in HPF or the condition flag of a processor in the Connection Machine, our system successfully optimizes the execution of conditional statements.

There are other massively parallel programming systems that are geared towards researchers. An example of such systems is Swarm [41]. However, the user of such systems needs to go through the edit-compile-test cycle to evaluate their simulations and has to write programs textually in Objective-C or Java.

As a side note, the implementation language of the first StarLogo was "*Lisp" [36]. It was simpler language compared to the later languages, yet it did require StarLogo to make the parallel programming accessible to the end-users.

The implementation language of a system should needs some consideration in this setting, where a world wide distribution is an important factor. One of the popular implementation languages of this kind of systems is Java. Java provides good performance in general cases but it is not suitable to implement specialized, vector operations that are needed in massively parallel simulations. Also, the static nature of the Java language makes it harder to implement a flexible object system. Finally, the platforms used in the major educational settings such as PDAs like Simputers and \$100 Laptop computers [42] won't support Java (or the full set of Java that is enough

to run these language systems implemented on top of it). Squeak is a more viable platform if we take such platforms into account.

With this observation, we have chosen Squeak as its implementation language, and use Squeak eToys as the base of Kedama.

2.3 The Overview of Squeak and eToys

The Kedama system is implemented on top of Squeak. The programming language of Squeak and its implementation have many interesting aspects that are useful for implementing a system like Kedama. In Squeak, there is an educational environment called Squeak eToys, or simply “eToys” implemented. Kedama is in turn implemented as an extension of eToys and inherits much of the characteristics of eToys. In this chapter, the overview of Squeak and Squeak eToys is explained. For more detailed description of them, including historical accounts and perspective, refer Appendix A.

2.3.1 Squeak: The Language

Squeak is an object-oriented, class-based, single inheritance, pure, dynamic programming language. The concepts of object-oriented, class-based, and single inheritance are more or less known concepts today. (Though, each different object model has slight twists on the definitions. The object model of Squeak is explained in Appendix A.) Below, the explanation of “pure” and “dynamic” aspects of the language is given.

The concept of “pure” means that everything in the language is an object, and all computation is specified in terms of objects and messages between objects. Most notably, the meta concepts in the language such as classes, the compiler, development tools like debugger, data structure for execution such as contexts (stack frames) are first class objects and accessible in the language. For example, defining a new class is specified as a message to the superclass with the specification of the new subclass as arguments. In a sense, the language is “reflective”, because the reified view of meta concepts of the language is available as objects in the language itself.

The concept of “dynamic” means that modification on the code is reflected immediately. An example is to add an instance variable to an existing class. When the request of adding an instance variable (in the form of a message to the class) is being processed, the instance variable is added to all the existing instances. As described in the following section, Squeak is not only a language but something that models an entire computer. Because of

this, the dynamic modification is essential part of the computation because the some persistent system objects needs to be modifiable.

2.3.2 Squeak: The Environment

Squeak (and Smalltalk) is also considered as an entire computing environment. This means that not only the language elements, but also the development tools and the (abstract view of) physical peripheral devices are provided as objects in the system. For example, the Browser tool, which is the standard tool to write program, and the Compiler that compiles the code to the executable code are accessible as objects.

The peripheral devices are also represented as objects in Squeak. For example, the keyboard and pointing device are represented as an object called **Sensor**. The input from the user is converted to **Event** objects, and such an event is sent from **Sensor** to other objects and processed by them. Also, there is an object called **Display** that represents the physical display screen. **Display** object is a kind of standard 2D graphics object in Squeak and data can be written to *and* read from it in the same manner as other graphics object. The written bits to **Display** are sent to the physical screen, so that the program can control every pixel on the screen.

The GUI frameworks that are used to develop its own tools are also written in Squeak itself. What a GUI framework does is to manage graphical objects and process various events. By using the objects such as **Sensor** and **Display**, the framework can dispatch events to the graphical objects, and the graphical objects compute their positions and appearances, and draw themselves on **Display** through an interface.

2.3.3 Squeak: The Execution System

Squeak employs the virtual machine (“VM”) and the virtual image (“VI”, or often simply called “image”) to execute the Squeak environment. The VM provides a well-defined interface that behaves as abstract hardware. On the virtual hardware, the binary image of the memory space (namely, the virtual image) is loaded and executed. The virtual machine has data that are necessary to execute the code in the virtual image, such the instruction pointer, stack pointer, and so on. Also, it provides the memory management system that allocates objects and reclaims unused ones.

The VI is populated with Squeak objects. Such objects include high-level objects for UI to the low-level ones that take the responsibility to interface with the VM. The executable code is byte-coded platform-independent se-

quence of instructions (*bytecodes*) and represented as objects. The *contexts* (stack frames) are also objects. An executable method is an instance of `CompiledMethod`. A `CompiledMethod` contains a sequence of bytecodes as well as some literal values. A context is an instance of `MethodContext`.

The instruction pointer in the VM points to a memory location in a `CompiledMethod`, and the stack pointer points to a location in a `MethodContext`. To carry out the computation, the VM fetches bytecode instructions, and mutate some slots of objects. Also, the VM accesses the actual platform (hardware and the host OS) API if necessary.

The image format is completely machine-independent. The object format only assumes the 32-bit (or 64-bit) linear address and 32-bit (or 64-bit) arithmetic. The linear chunk of data can be written out to a file and later can be loaded on to any correct Squeak VM on any platforms. The difference of little endian and big endian is absorbed upon loading a saved image onto a VM.

The Squeak image allows to have *homogeneous array* objects. A homogeneous array can contain uniform non-pointer data in the platform's native format. For example, a homogeneous array called `FloatArray` contains 32-bit `float` values that are in IEEE 754 format, and a `ByteArray` contains 8-bit unsigned integer values, etc. The content can be accessed from C-compiled code as `float[]` or `unsigned char[]` without any conversion so that the iteration over such arrays can be executed as fast as a program written in C.

The set of C-compiled code that either to call the platform dependent API or platform independent function to optimize loop intensive computation is called *primitives*. The implementation of VM and the platform independent primitives are written in Squeak itself, and translated to the C language. The platform C compiler then compiles the translated code and link with the platform dependent C functions. Since vast majority of code is platform independent, the Squeak VM is highly portable.

2.3.4 The Basic of eToys

The Squeak system itself is a general purpose programming environment. However, the original motivation to create Squeak was the desire to have a system for implementing an educational and media authoring environment for “children of all ages”.

EToys is one of such attempts. Its primary target age group is around 11 years to 12 years old, and its focus is on the interactive graphical programming environment. The user constructs a *script* to control graphical

objects by dragging and dropping the code fragments that are represented as graphical entities called *tiles*.

In eToys, the user constructs media-rich scripted contents by pointing and clicking, dragging and dropping, and typing parameters such as numbers and objects' names. To foster the constructivist theory discussed in Section 1.1, the ideas such as “incremental construction”, as well as the combination of visual representation and symbolic representation of objects are incorporated into the eToys' user experience. Incremental construction allows the user to explore the problem domain by making smaller changes and observe the feed back from the system. The visual representation of objects allows the user to relate himself to the user objects. At the same time, the symbolic representation of objects and code allows the user to write a well-defined program with them.

The origin of the system was to have something that is just good enough for demos, but not necessarily for the *real* end-users. However, a few brave teachers started using the system in the real classrooms with support from the Squeak team. This on-site testing over the years led the improvement of system that is stable enough for general public.

EToys is used as the basis of the Kedama system. In the remaining of this chapter, EToys is illustrated with figures, as well as the limitations of it that led us to the Kedama system.

The objects that the user manipulates and scripts in the eToys system are graphical entities that are visible and interactive. The user can instantiate the pre-defined prototypes such as Rectangle and Ellipse for basic shapes, Text for editing text, or make his own painting by using the Squeak's painting tool. When the painting is finished, the painting is automatically converted to a graphical object. Apart from these simple shapes and paintings, There are many more complex objects such as games and movie players. Note that an eToys object is in similar but at the same time very different object model from that of Squeak's. In the following the term “object” may refer to the eToys' object, but the disambiguation is straightforwardly given from the context.

A screenshot from a typical eToys session is shown in Figure 2.1. In the picture, there are two user drawn objects that look like a car and steering wheel, and an object that looks like a rectangle. Also the tools used to script objects called a *viewer* for the car object at the right edge are shown as well (these are explained later). The entire Squeak desktop is called the “World”.

There is a “meta-select” gesture, which is usually done by clicking an object with Alt-key or Command-key pressed. This gesture selects the object

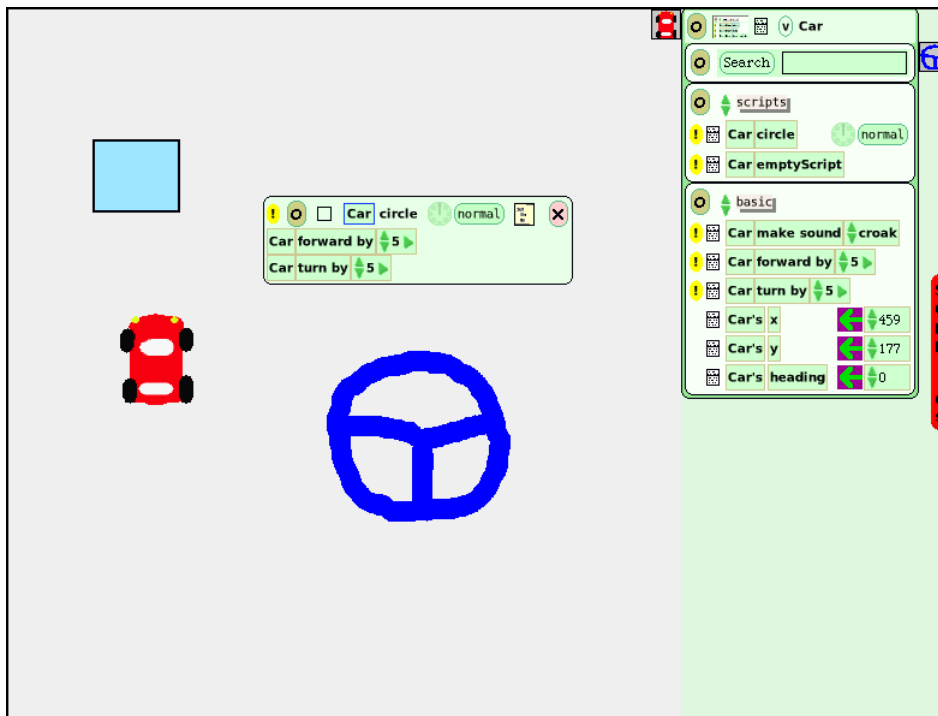


Figure 2.1: A screenshot of eToys session.

and brings up “halos” for the object. In the Figure 2.2, the car object has got the halos. In a sense, halos are graphical version of context menus [43] that provide context dependent features for the selected object. It is useful to avoid having stationary menu-bar or such. Examples of features include re-sizing, rotating, brings up the “property sheet” to change object’s properties such as colors and borders. For example, if the user drags the blue-colored halo at the bottom-left, the object rotates along with the drag-gesture.

When the user clicks on the turquoise-colored halo on the left of the target object, the viewer for the object is opened. A viewer is a tool to “look into” the object. Inside the viewer, the symbolic view of the target object is shown in the form of readouts for the properties. See Figure 2.1 again. There is an object that looks like a car, and the viewer of the car is clinging on the right edge of the screen. Some properties of an object are *intrinsic*; they are really owned by the object. The x and y coordinates and heading properties are the examples of intrinsic properties. On the other hand, “non-intrinsic” properties are defined in relationship with other

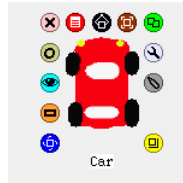


Figure 2.2: An object with halos.

objects, such as `isUnderMouse` (a Boolean-type property that shows if the mouse cursor is over the object).

Each object can have its own viewer, but the system shows only one viewer at a time. Other non-active viewers are retracted as if they were off the right edge of screen. When a non-active viewer becomes active, it is “pulled out” from the edge and the content becomes visible. The retracting and pulling-out action uses the analogy of a “flap” or a drawer. The knob of the viewer shows a thumbnail image of the viewed object. In the Figure 2.1, the viewer for the car object named `Car` is active and opened. The viewer for the steering wheel is closed and the knob is clinging at the right edge.

The viewer also holds the list of tiles that represent the commands for the object. The execution of a command typically causes some change to the program and object state. Let us call the dynamic effects caused by the execution of command *actions*. The eToys objects understand common commands, but different types of objects have idiosyncratic commands. For example, a text object has text-related commands, or a holder for other objects has collection manipulation commands. These idiosyncratic commands only show up in the viewer for the particular object.

A script is a visual object that holds the sequence of tiles. To edit a script, the user drags out the tile that represents a command from the viewer and drops it onto a script. A tile that represents an assignment for a property can be also dragged out from the viewer and dropped onto a script as well. The reference to a property can be also dragged out to be used in the expressions and an argument for a command. A property has pre-defined type. The user can drop a property onto a place in a script only when the type of the property matches.

A property reference is specified as a pair of the owner object and the name of the property, and shown with the possessive: “obj’s prop”. Refer to the viewer shown in Figure 2.1 for examples, such as `Car’s heading`.

The user can create and edit scripts by assembling the *tiles* that represent a) commands, b) the assignment to the properties of objects, and c) the

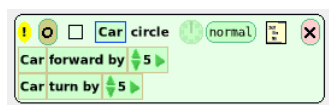


Figure 2.3: A script that makes the Car object go circle.

references to the properties of objects. When the user does the drag-out gesture for a variable or command from the viewer, a tile that represents it is created. The user can also create an empty script from the viewer. To edit the script, the user can add or remove a command, drop a variable onto the argument position of a command, change the numerical arguments, and expanding arithmetic expressions. The user can also specify the name of the script. Figure 2.3 shows a user-created script named “circle” with turn by 5 and forward by 5 commands.

A script can be marked as *ticking* by the user. A ticking script repeatedly executes itself at a regular time interval. (The status of the script such as ticking is controlled at the header part of the script explained below.) Multiple commands in a script are executed one by one from top to bottom. For example, the script in Figure 2.3 makes the car drive in a circle, because “it moves a little and turns a little over and over again”. Then user can mark multiple scripts as ticking. In this case, each script is executed atomically and provides a way to write concurrent programs.

Ticking is one of the central ideas in eToys. The argument for a command, such as “5” in “Car forward by 5” and “Car turn by 5”, essentially is a differential from the previous state. Then, ticking a script, or repeat the same command over and over again, means that it is effectively doing the integration of the differential. By letting the children focus on the derivative form that only uses additions such as “Car forward by 5” and “Car turn by 5”, and letting computer do the integration, the differential geometry can be presented to children in a understandable way (recall the quote from Bruner in Section 1.1). Such offset like “5”, as it is done in this circling car example, is specified in the local coordinate system of the car. One of the Papert’s powerful ideas was that the concept of local coordinate system matches well with the children’s perception of the world, where they are always at the center of the world.

The graphical representation of the script is called a “scriptor”¹. A scriptor (See Figure 2.3) has two parts. One is the header part that holds

¹The name may seem grammatically incorrect, but it is actually a short-hand form for “script editor”, according to the implementer of it.

some buttons to control the script. The other is the body of script. To the body of script, the user can drop command tiles to add them to the script. A statement in a script can be either:

- A command. It consists of a receiver, the name of the command, and zero or more arguments.
- Various forms of assignment. The left-hand side is a reference to an object's property. The right-hand side is an expression consisting of properties and literals. The assignment operator can be a simple assignment or updating assignments. An updating assignment is analogous to the `+=` and similar operators in the C programming language. For example, the "increase by" assignment updates the left-hand side by adding the original value and the result of right-hand side. There are "decrease by" (`-=`), and "multiply by" (`*=`), also.
- A Test-Yes-No statement. The Test clause is a Boolean-typed expression and the Yes and No clauses can contain other statements recursively.

The first example in Logo was a turtle making a circle. In eToys, it is a car making a circle. However, the biggest difference of eToys from Logo is that scripts can be modified at any moment, even when they are running (i.e., ticking). Any modification such as adding/removing a command or changing a parameter takes effect instantly. The dynamic nature of the system enables the user to quick exploration of the problem domain.

Now, let us consider not only to have a car going circle, but make it controllable by a steering wheel. The arguments for commands, "5" in "Car turn by 5" for instance, are editable by typing the number or clicking the spinners (up and down arrows) by it. The change of the parameter is instantly reflected and the curvature of the car's trail changes; it is a very rudimentary way but the user can drive the car.

However, it is often joked that driving a car by clicking buttons is like "kissing your sister"; it is not as exciting as the real thing. What we would like to have is another object that looks like a steering wheel, and the heading of the wheel affects the car's direction.

Here, the user can drag out the heading variable from the viewer for the object that looks like a steering wheel, and drop it onto the argument number for Car turn by command in the script. Once the tile is accepted as an argument, the heading value of the steering wheel is used as the turn by amount, so that the user can *really* drive a car by rotating the steering wheel via the blue halo. See the script in Figure 2.4 for this.

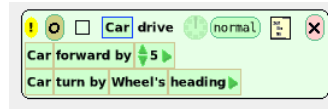


Figure 2.4: A script to drive the Car object with Wheel object’s heading.

This is the typical scenario in actual classrooms. It always turns out to be a very exciting moment for children when they realize that they can make stuff on computer do what they tell. For educators, it is a compelling argument that teaching differential geometry to children in an understandable way is possible.

The user-created content is called a “*Project*”. A Project can be saved into a file, and later loaded into another Squeak session. There are ways to share the project over the Internet.

The above gives the gist of the eToys system from the user’s point of view. As you may see, the term “object” in the eToys context refers to something different from Squeak’s context. The first difference that comes to the user’s mind would be that the eToys objects always have graphical appearance. More notably, however, the way that the specialization of objects in the system is organized differently. In one sense, an eToys object is hybrid of a class-based object and an instance-based object. Upon the initial creation, an instance (a rectangle, for instance) can be created from a pre-programmed class-like structure and later the instance can have its own shape and behavior by specializing the instance it self. One limitation of eToys is that the use of the specialization made by the user is limited to itself or its “siblings”. The user can’t have something analogous to subclassing; it is not possible to have another set of objects that borrow the specialization of parent and further add their own. This limitation is considered good, as the typical user program doesn’t require the nested specialization that adds too much invisible relationship between objects. Siblings are the only way to use the existing specialization on an object to other objects. An object can create the similar objects that share the same behavior and shape. (These are called siblings.) The modification of the behavior or shape through one of the siblings affects others, as they always share the same structure that represents the shape and behavior (in fact, a Squeak class).

As described in Section 2.3.5, eToys has its limitations, and the user often hits the limitation quite early, when he tries to control dozens, not to mention hundreds of objects. The Kedama system tries to provide an “upgrade path” to such a user. In the next chapter, the extended user

experience by the Kedama system is illustrated.

2.3.5 From eToys to Kedama

While eToys has been successful for its target age group, it has its limitations and drawbacks. EToys attempts to provide the programming environment that involves a few concrete objects that the children can relate themselves to (i.e., a child writes program for an object in the manner of “what I would do”). However, for older children, a program with more objects, and more importantly, with more *abstraction* is required. As written in the introduction, a complex system simulation with massively parallel objects is very effective tool to model various phenomena but the current eToys is not suitable for the purpose.

From the implementation stand point, the computational cost for transforming (e.g., moving, rotating, scaling, etc.) the graphics becomes the problem when there are massive numbers of objects. The user interaction becomes sluggish when dozens of objects are moved by some ticking scripts. To write a simulation of a complex system, it is essential to have better performance.

Chapter 3

The Overview of The Kedama System

Kedama is a programming system for massively parallel objects that is built as an extension of eToys. Kedama's is to provide a usable system for real users, and there are a few decisions to be made in the design of UI. Specifically, keeping the interactive nature and familiarity with the eToys interface are the key issues.

This chapter explains the user experience of the Kedama system. The new kind of objects added to the system and how these are used by the user are illustrated with a series of user interactions and responses from the system. For more detailed specification of the new objects and their commands and slots, refer to the Appendix B.

3.1 Objects in Kedama

The Kedama system defines three new kinds of eToys objects. The one that plays the central role is a *parallel turtle*. A group of homogeneous parallel turtles is called "a breed of turtle". Another object is called a *Kedama World* that represents a place in which the turtles and patch variables (explained below) reside. A Kedama World is a two-dimensional plane and provides the coordinates and headings for the turtles and patch variables in it. By default, the logical extent of a Kedama World is 100x100. The last type of object is the *patch variable*. A patch variable provides a two-dimensional matrix of cells. Each cell of this matrix corresponds to a grid point in the coordinates of a Kedama World and holds an integral value. In another view, the cells serve as environment and communication medium for turtles. The turtles

communicate with each other by changing and reading the environment.

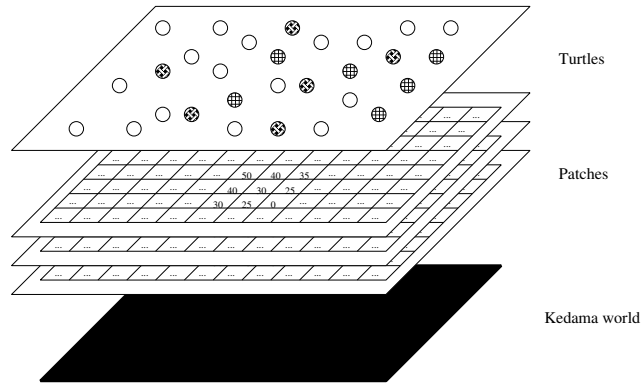


Figure 3.1: The schematic view of the system.

A state of a running Kedama program can be depicted as a schematic figure in Figure 3.1. At the bottom, a Kedama World sits and holds the other objects in it. On the Kedama World, there are patch variables. Upon rendering a patch variable to graphics, the values in the cells are converted by a function from integer to color. The picture also shows that there can be more than one patch variable in the system. There are turtles over them. In the schematic figure, colored small circles represent turtles. In this schematic view, the breed (the group of turtles) is not shown, but the group of same color turtles represents a breed. Again, the Figure shows that the system can accommodate more than one breed of turtle; in this figure, there are gray breed and white breed. (In the actual system, the turtles in a breed can have different colors, as the color is a property owned by individual turtles.)

On the actual eToys screen, these Kedama objects are presented to the user as graphical representations. A Kedama world is typically magnified and shown as a bigger square than its logical size (because 100x100 is a tad small to see a simulation in it). Turtles in the Kedama World are rendered as colored pixels. The graphical representation of a patch variable is also a square, but usually shown without magnification. Each pixel in the graphical representation of a patch variable is a rendering of the integral value at the corresponding patch cell. A breed has a graphical representation that is a colored square. The graphical representation is also called an *exemplar* of the breed.

What makes an eToys object an eToys object is the ability to open the viewer. The exemplar is provided to be an eToys object for the user for two

reasons. One is that the user needs to be able to open the viewer for an object to interact with; individual turtle is rendered as a dot and hard to click on. Secondly, the programming style in Kedama is to send a command to a breed of turtle, and all turtles in the breed follow the command; an entity that represents the breed is needed. In other words, the user doesn't use the name or the reference to individual turtle in scripts. Rather, he only uses the name of a breed in the program, and the turtles in the breed perform the actions. The Kedama World and the patch variables are valid eToys objects without limitations.

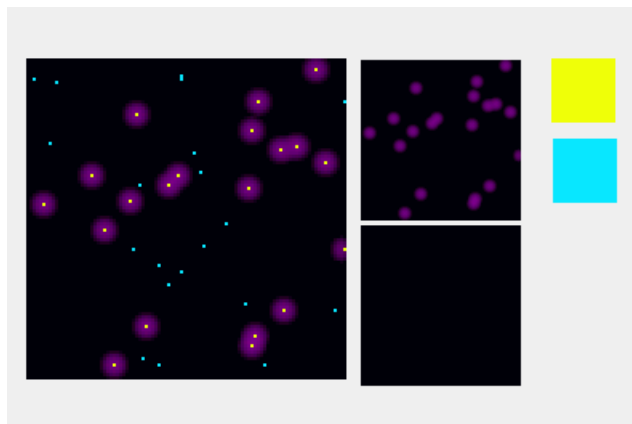


Figure 3.2: The graphical representations of a Kedama World (left), two patch variables (middle), and two turtle exemplars (right).

Figure 3.2 shows five objects of the three types. The visual look of the Kedama World (shown at left) shows all other entities in it. At the middle in the figure, two patch variables are shown. The Kedama World is magnified by factor of two, so it is shown twice as higher and wider than the appearance of patch variables. Also, two exemplars for turtles are shown at the right.

These three types of objects can be manipulated in the same manner as standard eToys. They responds to the “meta select” action (Alt + click) to get the halos and viewers, and the commands and properties specific to each new object can be used in scripts.

To disambiguate the terminology, the desktop, or the World for an entire Squeak project is called an “eToys World”. A Kedama World is always referred to as such.

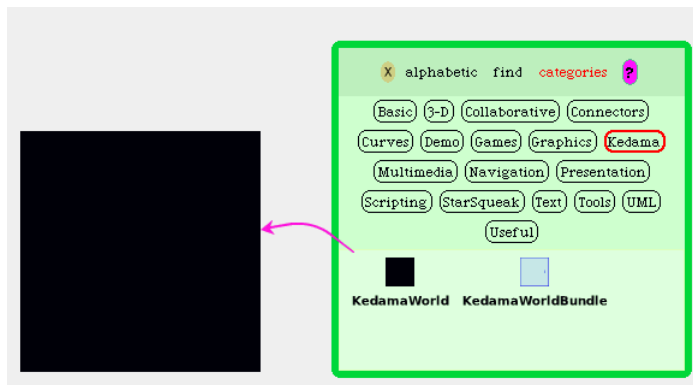


Figure 3.3: Instantiating a Kedama World from the standard Object Catalog tool.

3.2 Setting Up Objects

Here, we follow a typical usage of the Kedama system. To get started, the user instantiates a Kedama World object from the standard tool called “Object Catalog” that lists a variety of user-accessible objects. (See Figure 3.3).

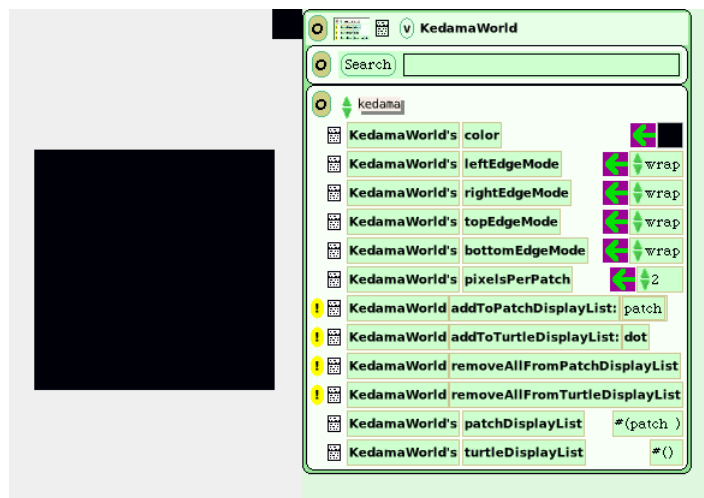


Figure 3.4: The viewer for the Kedama World object.

In the viewer of the Kedama World, there is a category called “kedama”, which contains the Kedama World specific properties and commands. Figure

3.4 shows the Kedama World object and its viewer that is showing the “kedama” viewer category. (Table B.1 in the appendix lists the Kedama World specific properties and commands.) Some of the properties have both getter and setter to allow reading and writing, while the others are read-only and only have getters.

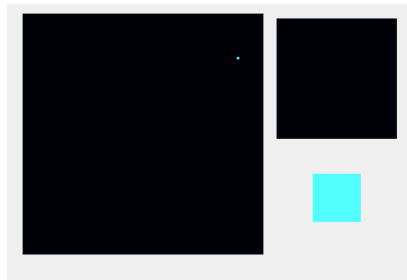


Figure 3.5: A Kedama World, a patch variable, and an exemplar of a breed.

From the viewer’s menu, the user can create a new patch variable and a new breed. When the user creates a breed, one turtle in the breed is also created in the Kedama World. Figure 3.5 shows a Kedama World, a patch variable, an exemplar of a breed, and the turtle in the breed in the Kedama World as a light blue dot. The breed is named “breed1” and the patch variable is named “patch1” in the following.

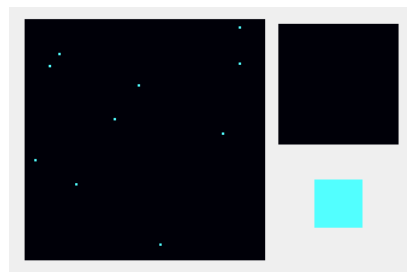


Figure 3.6: A Kedama World with ten turtles of a breed.

3.3 Turtle and Breed

Here, the user can open the viewer for the exemplar of `breed1` that has a property called `turtleCount`. When the user changes the value of the property, the number of turtles in the breed changes accordingly. For example, if the

user sets the value to 10 from 1, 9 more turtles are created and placed at random positions. With these 10 turtles, the Kedama World will look like Figure 3.6. (Also, refer to Table B.1 in the appendix for the turtle specific properties and commands.)

One thing to note is that the basic properties of a turtle, such as x , y and heading are represented as floating point numbers internally. These values are rounded to integral values when needed.

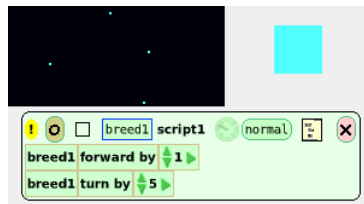


Figure 3.7: A script with forward by and turn by command.

Let us create a script that looks like Figure 3.7. The textual equivalent of the script is as follows. The following textual form will be used in the rest of dissertation interchangeably with the graphical representation.

```
script1
  breed1 forward by 1
  breed1 turn by 5
```

What will happen when we execute this script, when the receiver of the message is an exemplar? When a command is sent to an exemplar, the action is performed by all the turtles in the breed. For example, when the above `script1` is executed, all turtles in `breed1` move by 1 to their forward directions and then turn by 5 degrees.

What happens when the user executes the script and a turtle is going off an edge? The user can specify the behavior when a turtle hits an edge, but the default is to move the turtle to the opposite edge, or to introduce torus like topology to the Kedama World.

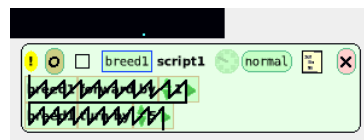


Figure 3.8: The execution flow in `script1`. (See Chapter 5 for detail.)

The reader may wonder about the order of action in this script, as many turtles execute the same commands “almost at the same time”. The detail will be discussed further in Chapter 5 but, in this chapter, only the following two points is enough to know; one is that all turtles in a breed execute a command before they execute the subsequent command. All turtles execute the command one by one so that the effects from one turtle are visible to others. The second is that there is only one thread of execution per a script invocation; the thread runs through a script when the script is invoked. By applying these two rules, the flow of control is as follows in this example; the thread first iterates over the turtles in `breed1` and executes `forward` by action on each turtle. Then, the thread advances to the next command and iterates over the turtles, and executes `turn by` action on each turtle.

Figure 3.8 depicts the flow of control as the black segmented line. Imagine that the turtles are lined up horizontally on the script, and each vertical line segment shows the execution of the command on a particular turtle. The zigzag line shows the same command is executed on individual turtles in the breed.

3.4 Patch Variables

A turtle has a special property called `patchValueIn`. This property is non-intrinsic (see 2.3.4), and takes one extra argument of patch variable-type. It accesses the value of the cell at the grid-point where the turtle resides (i.e., the x and y coordinates of a turtle are rounded to access the patch cell). For example, a script with two assignments:

```
breed1's patchValueIn patch1 ← 10
breed1's x ← breed1 patchValueIn patch1
```

where `breed1` refers to a breed and `patch1` refers to a patch variable. When this script is invoked, it will first store 10 to `patch1`'s patch cells where the turtles in `breed1` reside. Then, the same values are read by the right-hand side of the second assignment (`breed1's patchValueIn patch1`). The same values are read because the turtles don't move between the first statement and the second statement. And, the values are stored to the x property of the turtles.

A patch variable understands a command called `diffusePatchVariable` used in Figure 3.9. When it is executed, a value in each patch cell is “diffused” to the neighboring cells; in other words, for each cell, the average of the content in itself and eight neighbors' content is stored as its new value.

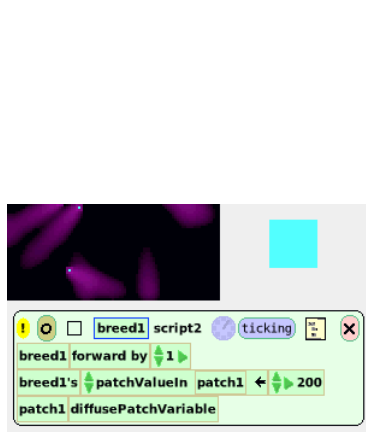


Figure 3.9: The tile implementation of script2.

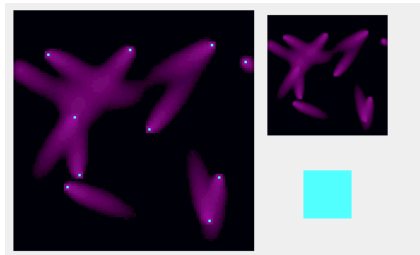


Figure 3.10: A fun Kedama project that puts meteor-like tails to the turtles.

With `patchValueIn` and `diffusePatchVariable`, a fun graphic effect can be written. Let us think a script with three steps:

1. The turtles keep moving.
2. The turtles store 100 at their position in `patch1`.
3. The values in `patch1` are diffused.

This script can be straightforwardly implemented with a script shown in Figure 3.9 or its equivalent textual notation:

```
script2
  breed1 forward by 1
  breed1's patchValueIn patch1 ← 200
  patch1 diffusePatchVariable
```

If the script is ticking, it creates a meteor-like visual effect as shown in Figure 3.10.

Note that the same execution order applies even when a non-turtle object is mixed in a script. When `script2` is executed, the first two statements are executed for all turtles in `breed1`, and then the third statement `patch1 diffusePatchVariable` is executed *just once*, because it is not a command for



Figure 3.11: The execution flow in script2.

turtles. Figure 3.11 depicts the flow of control with a black line segments. Notice that the thread doesn't show the zigzag segments for the third statement; this means the command is executed just once per invocation of the script.

3.5 Collision Detection

A typical function that one would want to have in a particle system is collision detection. In Kedama, collision detection can be written by the user himself by using a patch variable.

A script in Figure 3.12 or its textual equivalent:

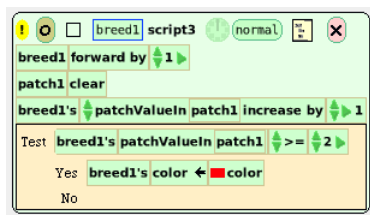


Figure 3.12: The tile implementation of script3.

```

script3
  breed1 forward by 1
  patch1 clear
  breed1's patchValueIn patch1 increase by 1
  Test (breed1's patchValueIn patch1) >= 2
  Yes
    breed1's color ← Color red

```

specifies:

- The turtles keep moving.

- Clear all cells of patch1 to zero.
- The turtles increase the patch cells at their position in patch1 by 1. After completing above, the values in a patch cell is equal to the number of turtles on the grid.
- Each turtle checks if there are 2 or more turtles at the grid, and if so, it changes its color to red.

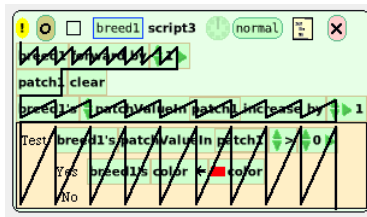


Figure 3.13: The execution flow in script3.

The execution order of this script again follows the same model (see Figure 3.13), and it gives the expected result. A turtle that collides, or occupies the same grid as others, changes its color to red.

Note that the actions caused by a statement are executed on each turtle in the breed one-by-one and the results from one turtle are visible to others. In this model, the statement successfully accumulates the number of turtles at the same grid point into the cell values, rather than *always* store 1 to the patch cell.

Another important idea of a patch variable is that it decouples the turtles; the turtles don't have to have the direct reference to others. It is usually a good idea to avoid direct references between turtles whenever possible.

The above illustrates the basic features of Kedama and their usage in a few scripts. While the feature set used above may look limited, there are a lot of examples that can be constructed from the above features as their building blocks.

In the next Chapter 4, a few notable examples are explained.

Chapter 4

Examples in Kedama

In this chapter, three notable examples of Kedama will be explained. The first one is a simulation of epidemic disease transmission. In this simulation, a village where a number of people live in is modeled. Then, one of the villagers gets a kind of contagious disease, and transmits it to a healthy other villagers when they collide. By changing the number of villagers or the contagiousness of disease, the user can learn the characteristics of different disease.

The second example is a simulation of ideal gas. In this example, molecules in a tank are simulated. One wall of the tank is movable and when a molecule hits the wall, some momentum is given to the wall. In this basic model, there is a wide design space to explore.

The third example is a simulation of forest fire. Its model is a field with trees, and some burning trees spread the fire to the neighbor trees. In a sense, it is similar to the epidemic example, but the trees don't move in this example. Here a feature to "fill" a Kedama World with turtles, or create a turtle at each grid point in the Kedama World is introduced. By writing a simple rule that involves a local condition (i.e., when a tree starts burning), the simulation exhibits the collective behavior of turtles. This example also illustrate the case where the state of a step of the simulation is preserved so that the state of the next step can be calculated independently from it.

All of these examples, and many more are available at the official Squeak web site at: <http://www.squeakland.org>.

4.1 Epidemic Simulation

The simulation explained in this section is an epidemic disease transmission. In this simulation, a village where numbers of people live in is modeled and how the disease spread out to the villagers. We start with only one villager infected and infected villagers transmit the disease to healthy other villagers when they collide (and the newly infected villagers do the same). By changing the number of villagers and/or the “contagiousness” of disease, the user can learn the characteristics of different disease.

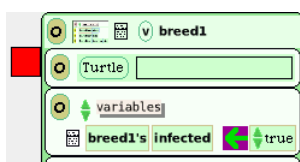


Figure 4.1: The infected property of Boolean type in the exemplar’s viewer.

In this example, we need each turtle (that represents a villager) to have a Boolean-type property called `infected` that indicates the villager is infected or not. The user can simply add a property to a breed from the viewer menu of the breed, just in the same way to do so for an ordinary eToys object (this would create a property with default name `var1` with `Number` type), change the type of the property to Boolean, and rename it to “`infected`”. As a result, the property will be shown in the viewer as Figure 4.1. This is the property owned by each turtle; there will be the number of instances in the system.

Also, to monitor the number of infected villagers, we add a property named “`infectedCount`” to the Kedama World. This is a kind of global property, and there is only one instance of it.

A typical simulation has “`setup`” script that initializes the state of simulation, and “`oneStep`” script that is repetitively executed to compute the next state. In this example, what the set up script should do is:

1. Set the number of villagers. Let us use 1,000 for the initial value.
2. Set the `infected` property of all villagers to `false` to make no villager infected initially.
3. Set `infectedCount` to zero to indicate all villagers are healthy.
4. Set the color of all villagers to blue to indicate they are healthy.

- Now, pick one villager and make him infected. The way we do this is to iterate over each villager and try to make him infected only if there is no infected villagers.

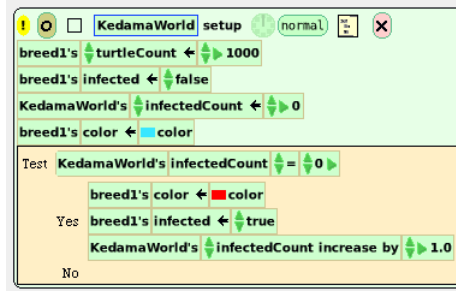


Figure 4.2: The setup script for epidemic simulation.

Figure 4.2 provides the implementation of this. Except the last conditional statement, it is written straightforwardly. As described in Chapter 3, a Test statement is treated as a big but single statement and it is iterated over the turtles. In this case, the value of infectedCount property is checked in the Test expression, and the value is incremented in the body of the statement. Because the value of infectedCount is zero at first, the Yes clause is taken for the first turtle. For the second and later turtles, the infectedCount is one so that the Yes clause is ignored. As the end result, only one turtle would execute the Yes clause.

In the oneStep script, what to do is:

- Move all villagers.
- Do the collision detection between infected villagers and uninfected villagers.
- Make all uninfected villagers that collide with infected ones infected.

The collision detection is similar to the example in Section 3.5. Here again, a patch variable is used a kind of map to record the presence of infected villagers. Then, the map is checked by the uninfected villagers to see if an infected villager is located at the same gridded position.

An uninfected villager gets infected when more than zero infected villager is co-located but the exact number of them is irrelevant; i.e., there is no need to use increase by. In this example, 10 is used as the value stored into the patch variable for the further modifications described below. The part that

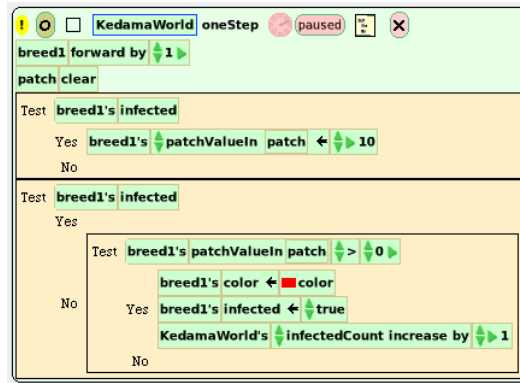


Figure 4.3: The oneStep script of the epidemic simulation.

does the collision detection is implemented as the second and third statement in Figure 4.3.

Then, each turtle checks whether : a) it is uninfected and b) the patch cell at its position has non-zero value. Because current eToys doesn't have the Boolean operations, we use nested conditional statements. In the body, we put the same three lines that makes the initial infected villager in the setup script.

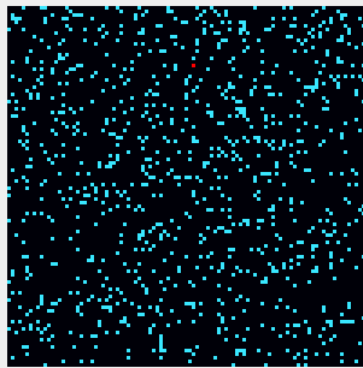


Figure 4.4: The initial state of the epidemic simulation.

These two scripts are the all we need. First, let us execute the setup script once to initialize the simulation. The initial state of the Kedama World will look like Figure 4.4. Notice that we have one red (infected) villager and many blue (999 uninfected) villagers.

Let us start ticking the oneStep script. The rate of the infection is slow

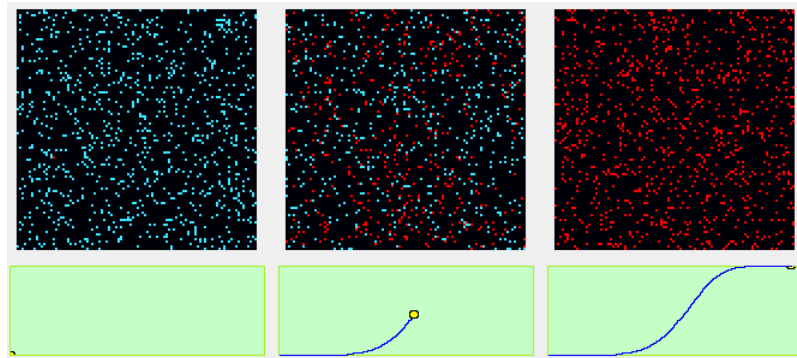


Figure 4.5: The progression of the simulation.

at first, but it becomes very fast at the middle, and then slows down when almost everybody is infected. In Figure 4.5, three different moments in the simulation are shown with graphs that visualize the number of infected villagers (these graphs are explained below). This is the basic of the epidemic simulation.

The graphs are drawn by a user object that the user can create and script in the standard eToys mechanism. (This is a powerful notion of eToys in which tools such as a graph plotter can be created by the user.) To make a graph plotter, we instantiate a small Ellipse (a standard eToys object), and set its `penDown` property to true. When the `penDown` property is true, the object leaves pen trail when it moves. Let us name the Ellipse “plotter”.

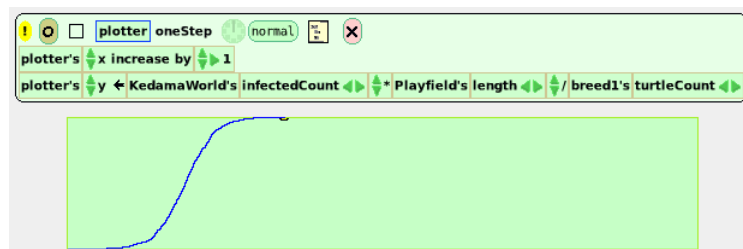


Figure 4.6: The plotter's script and the resulting graph.

For each step, what plotter should do is:

- Increase its x coordinate by 1.
- Set its y coordinate a value proportional to `infectedCount`.

By implementing these, the plotting script and the result will look like Figure 4.6.

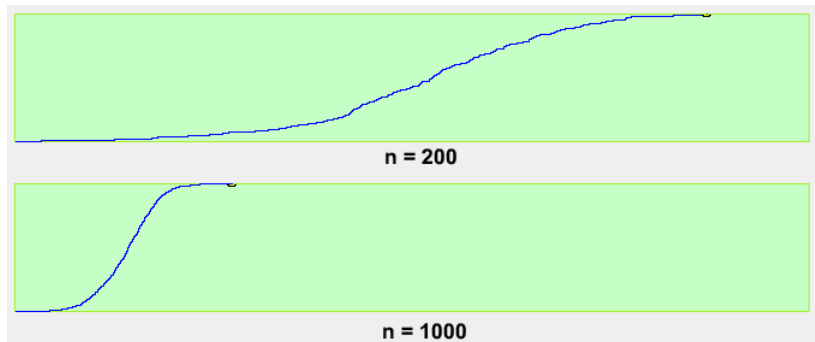


Figure 4.7: The graphs for 200 villagers and 1,000 villagers.

With this graph feature, we can compare the behavior from different parameters and code. As the first variation, let us change the number of villagers. If we reduce the number to 200, it takes much longer to infect all villagers, and the graph is stretched. In Figure 4.7, the below and above are the graphs with 1,000 and 200 villagers, respectively.

The other experiment is to change the contagiousness of the disease. If we put `patch1 diffusePatchVariable` (see Section 3.4), it effectively sets a positive value to the 8 neighbors around and the position of each infected villager. (I.e., it stores a value in a cell and then average the value with its 8 neighbors and store the value into the 9 cells. Since a patch variable can only store integers, the original number should be bigger than 8.) With this modification, an infected villager can infect not only the healthy ones at the same grid, but also around it.

Here we plot the result of simulations with 200 villagers with and without the `patch1 diffusePatchVariable` statement. In Figure 4.8, the above is without it and the below is with it.

One of the interesting educational value of this comparison is as follows; human's perspective on the historical progress tends to be narrow. To visualize the narrowness, we gray out a area of graph beyond the human's imagination and only show the narrow area clearly (See Figure 4.9). If we set our perspective at the similar point in the two graphs as Figure 4.9, we see a steep slope in the below one, but hardly any slope in the above one. This means that while the rate of infection for some kind of disease is slow (like AIDS), It can be equally dangerous. We need to understand the limitation on our perception and consciously try to extend it with tools like

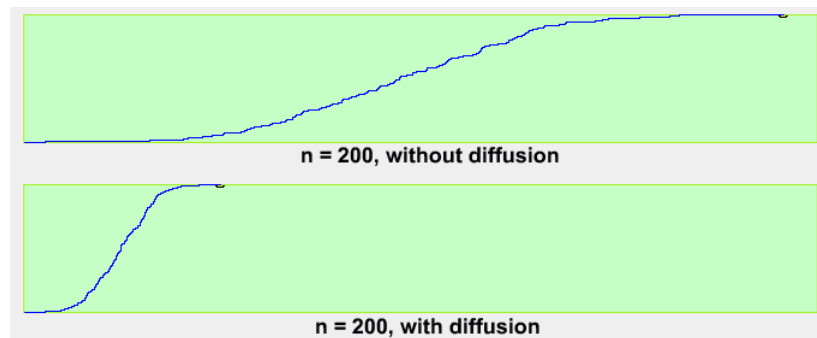


Figure 4.8: The graphs for 200 villagers with weak contagiousness and strong contagiousness.

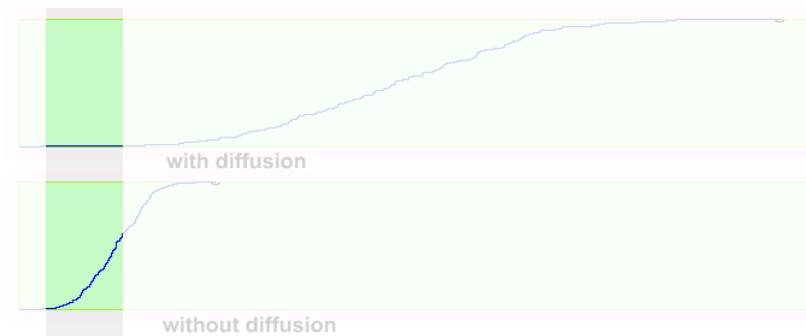


Figure 4.9: An illustration of the narrow perspective. Through this time window, it is hard to tell what is going on.

science.

The nested conditional statements is more complex than they need to be. This is a limitation of eToys where it doesn't have the logical-and and -or operations. However, other than that, the script is written in a straightforward way.

4.2 Ideal Gas Tank Simulation

In this simulation, molecules in a tank are modeled as turtles. One wall of the tank moves. And, when a molecule hits the wall, some momentum is given to the wall. Let us start from an empty Kedama World.

There are two new features used in the example. One is to set the “edge modes” of a Kedama World. The edge modes control the behavior when

a turtle hits edges of the Kedama World. The options for the edge mode when a turtle hits an edge are:

wrap the turtle shows up from the opposite edge.

stick the turtle is located at the closest possible position to the edge.

bounce the turtle bounces back.

The edge mode can be set for each of four edges individually.

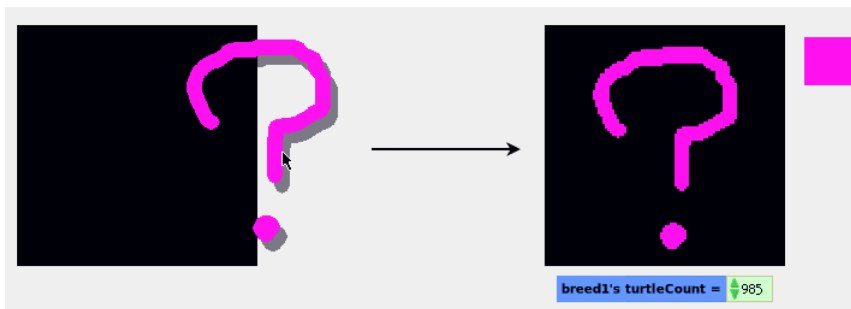


Figure 4.10: Dropping a Sketch onto a Kedama World creates turtles lined up with the bitmap.

Another feature used in this example is to create a breed of turtle by dropping a “sketch” on to a Kedama World. A Sketch is a graphical bitmap object in eToys and can be created in various means including by drawing a picture with the paint tool or importing a bitmap file from the platform. If the user drops a Sketch onto a Kedama World, turtles are created at the grid positions where pixels of the sketch are located. The created turtles get the color of the pixels of the sketch. For example, if we drop a sketch that looks like the one in Figure 4.10 onto the Kedama World in the Figure, it will create turtles of a new breed as shown in the Figure.

There are two “pit holes” in the implementation of this gas tank simulation. First, we will be using two breeds of turtle. One breed represents the molecules of the gas, and another represents the moving wall of the gas tank. Secondly, the positive y-coordinate direction of the Kedama World is downwards. It affects the sign of offsets in the simulation.

For the gas tank simulation, we draw a straight horizontal line in the painting tool, and drop it to a new Kedama World. After this, the Kedama World will look like Figure 4.11. The line that consists of turtles represents the movable ceiling. Let us name the breed of this turtles “ceiling”. Let us

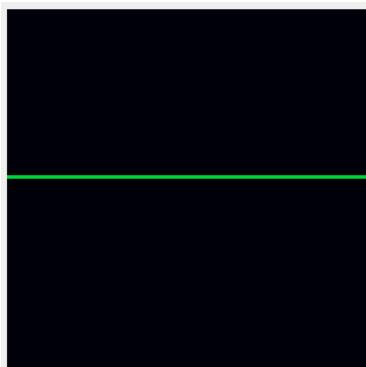


Figure 4.11: Create the ceiling of the simulated gas tank from a sketch.

consider the area surrounded by the left, right, bottom edges and the ceiling is the gas tank.

We then make the molecules move in the chamber. When a molecule hits a non-moving wall, it simply bounces back, but when it hits the ceiling, it gives small momentum to the ceiling and bounces back. At the same time, we add gravity that pulls the ceiling.

Let us make the molecule breed from the viewer menu of the Kedama World. For the later use, we also add a few properties to the Kedama World. These are `ceilingSpeed`, `ceilingPos`, and `gravity`. (Note that these are scalar values because they are properties of a non-turtle object). The `ceilingSpeed` property holds the current speed of ceiling. The `ceilingPos` property holds the intermediate value for the calculation of the y coordinate of ceiling for the next step.

Again, we will create a setup script called `setup` and a main script called `oneStep`.

What `setup` should do is:

1. Set `turtleCount` property for molecules. Let us use 2,000 for the initial value.
2. Setup the default values. Set the `ceilingPos` and y coordinate of ceiling to 50, `gravity` to 12, the edge mode for the top edge to `wrap`, and the edge mode for the bottom, left and right edges to `bounce`. (The top edge mode may be `bounce`, but `wrap` is used to get an interesting effect when the ceiling, which consists of a breed of turtle, goes off the top edge.)

The implementation of `setup` looks like Figure 4.12.

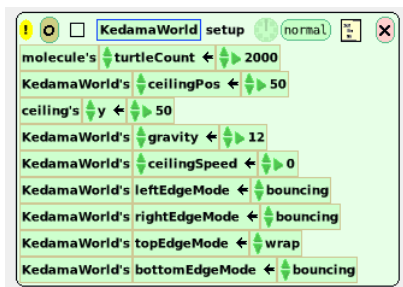


Figure 4.12: The setup script for the gas tank simulation.

What oneStep does is:

1. Move all molecules.
2. If a molecule hits the ceiling (i.e., move to a position where its y coordinate is less than the ceiling's y coordinate), increment ceilingSpeed and change the molecule's heading. Repeat this step for all molecules.
3. Decrease ceilingSpeed by gravity.
4. Decrease ceilingPos by ceilingSpeed.
5. Assign ceilingPos to the y coordinate of ceiling turtles.

The implementation of oneStep looks like Figure 4.13.

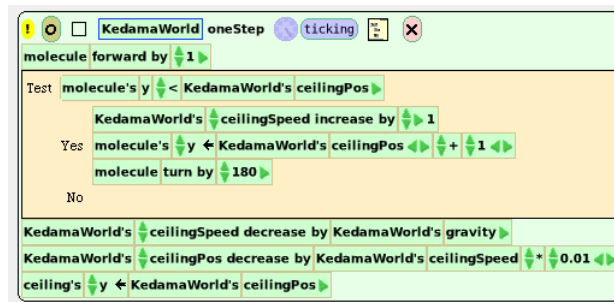


Figure 4.13: The oneStep script for the gas tank simulation.

It is often necessary to adjust the parameters arbitrarily in this kind of simulation. In this example, the speed is set to 1 for all molecules, the momentum given to the ceiling is also a constant (1), and the coefficient for the speed being added to the position is arbitrarily set to 0.01 (to smooth

the movement of the ceiling). According to the thermo dynamics theory, actual distribution of the speed of molecules follows Maxwell distribution. However, to model the collective behavior, we can just pick the average speed and use it for all molecules. The momentum a molecule gives to the wall is proportional to the speed of molecule (and the coefficient is a function of the mass and the size of the wall). Following the same simplification, we make it 1 as well.

Also, the bounce algorithm from the wall is simplified; calculating the bouncing movement for two moving turtles is not easy in general cases. Instead, the solution we take here is to make a molecule a 180 turn. This makes somewhat incorrect in strict sense, as a molecule that hits the ceiling from bottom-left to top-right goes back to bottom-left, not bottom-right. However, when we average the behavior of massive number of molecules, the x component of the movement can be abstracted out. Also, to keep the molecule below the ceiling, when it hits the ceiling, time, we add 1 to the y coordinate of the molecule.

The biggest simplification made here is, perhaps, the projection from 3D physical model to 2D implementation. With above simplification, the z component of the molecules are simply canceled out, and doesn't contribute to the change to the ceiling speed.

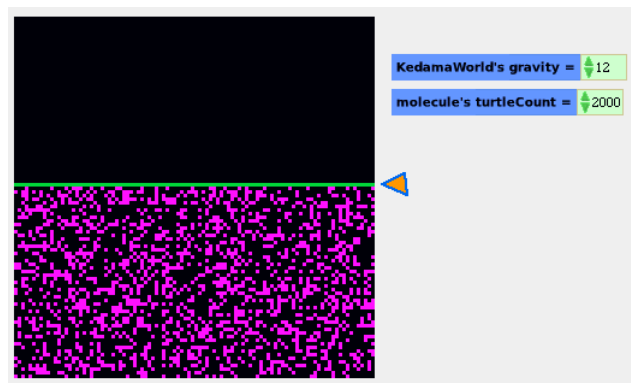


Figure 4.14: The equilibrium with 2,000 molecules.

If we run the `setup` script once, and tick the `oneStep` script, the ceiling becomes unstable at the beginning. However, after a short run, the system reaches the equilibrium (Figure 4.14). The triangle on the right indicates the rough position of the ceiling at this equilibrium.

Now, let us interact with the simulation by changing the program and parameters. To modify a property of an object, grab the “watchers” for

gravity of the Kedama World and turtleCount of molecule. A watcher lets the user see the value of property, as well as modify its value.

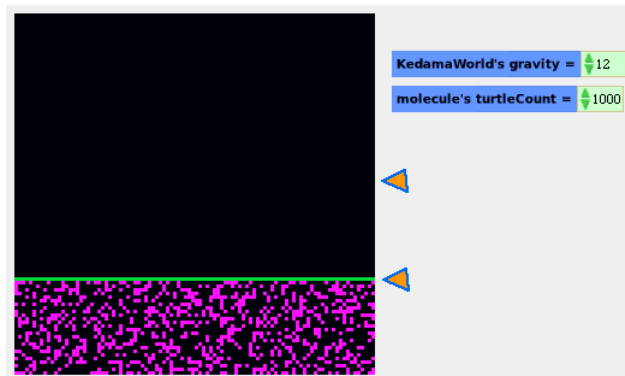


Figure 4.15: The equilibrium with 1,000 molecules.

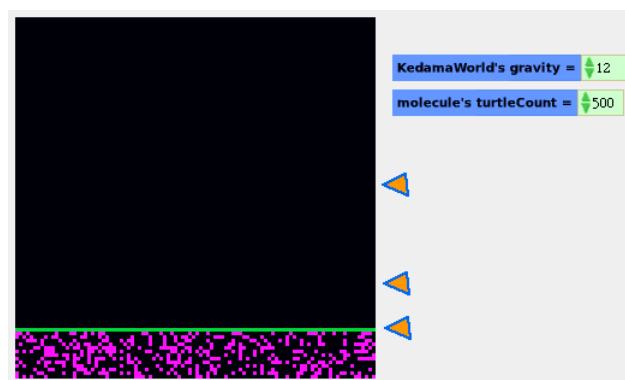


Figure 4.16: The equilibrium with 500 molecules.

Suppose the height of the tank at the equilibrium is around 52 pixels (from the bottom). If we cut the number of molecules half (to 1,000), the height at the equilibrium *gradually* moves to around 26 pixel from the bottom (See Figure 4.15). We can further cut the number half (to 500) (See Figure 4.16). Notice the turtleCount number in both figures.

What will happen if we return to 2,000 molecules from 500? This means that the number of molecules in the chamber increases by a factor of 4 *instantly*. It should cause some sort of explosion, and the simulation surely exhibits it.

We can change other parameters to see how the system reacts to the

change. Changes to gravity, molecules' speed, the mass (the momentum) all rightly follows the theory. With a dozens of lines of code that a student can write by his own, an interactive model which is reasonably good match with the reality. Learning a concept by constructing the model is the heart of the constructivist approach, and this example successfully shows it.

4.3 Forest Fire Simulation

The example in this section is a simulation of forest fire. This is a re-implementation of the same example in [16]. Its model can be considered to be a cellular automaton. A gridded space (represents a forest) is randomly *filled* with “on-” cells and “off-” cells initially. An on-cell represents a tree at the grid, and an off-cell represents an empty grid. Some trees are initially marked as “burning”. The rule for the cellular automaton is that if a tree has a burning tree at its “four neighbors” (the next grid points that is top, bottom, left, and right of the cell), the tree starts burning.

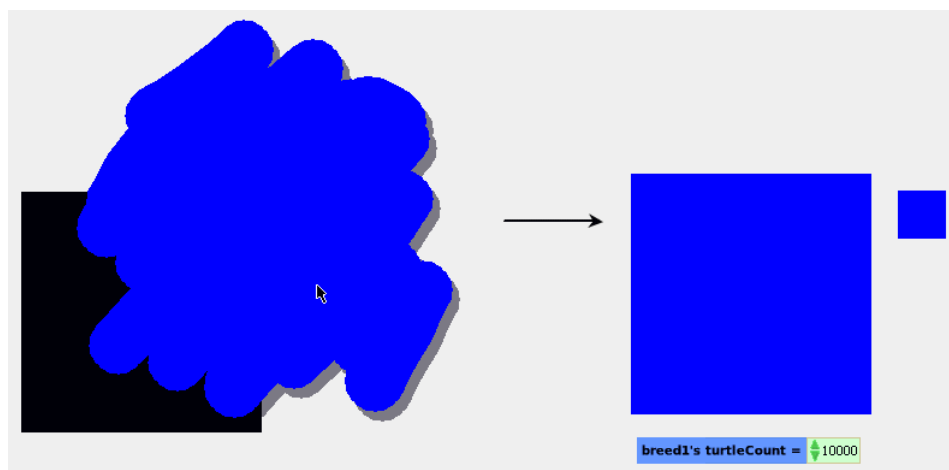


Figure 4.17: Fill a Kedama World with turtles by dropping a big sketch.

To make a cellular automaton simulation, we again use the feature to create turtles by dropping a sketch. In this example, we would like to “fill” all the grid points, and to do this, we draw a sketch bigger than the Kedama World, and drop the sketch so that the sketch covers the entire Kedama World. (See Figure 4.17.) As a result, A breed with 10,000 turtles is created, and the turtles are located at all grid points in the Kedama World. Let us name the breed `breed1`. The turtles will be made invisible, but will be doing

the calculation behind the scenes.

Here, we use two patch variables. One represents the strength of flame and is named `flame`, and the other represents the trees and is named `tree`.

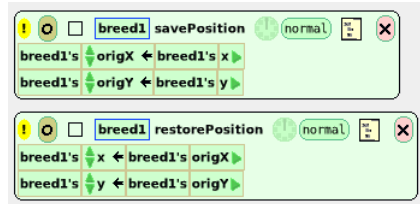


Figure 4.18: `savePosition` and `restorePosition` for a cellular automata type simulation.

The basic technique to implement a cellular automaton is to move the turtles around their “home positions”, let them do the operations on the neighboring patch cells, and get the turtles back to the home positions. To return the turtles to the home positions efficiently, we make each turtle remember its home position. For this purpose, we add `origX` and `origY` properties to `breed1`. Then, we write two scripts called `savePosition` and `restorePosition` that are shown in Figure 4.18. Execute the `savePosition` script once to remember the original grid point to these properties. Also, each turtle carries a property called `flameLevel` that serves as the cache or temporary variable used for calculating the state of the next generation.

There are several small scripts that serve as subroutines to other scripts. However, the core of the simulation is written in two scripts. In the following, these two are explained. The actual project, along with others, are available online at the official Squeakland site.

One of the important scripts is named `calcFlameLevel`. What it does is:

1. Initialize the turtles’ `flameLevel` property to zero.
2. Move all turtles to four neighbor positions, and accumulate the values in the `flame` into the `flameLevel` property.
3. Move all turtles back to the original position.

The assumption here is that before the invocation of `calcFlameLevel`, the other subroutines set the values in patch variable `flame` correctly. After executing this script, a turtle’s `flameLevel` holds the sum of the values of four neighbors in the `flame` patch variable. I.e., it indicates that any of the four neighbors are burning or not.

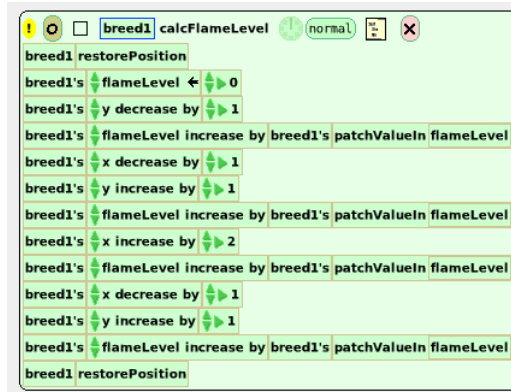


Figure 4.19: The calcFlamelevel script for the forest fire simulation.

The tile implementation of the method is shown in Figure 4.19. Because there is no built-in mechanism to visit four neighbors unlike StarLogo or NetLogo, The movement needs to be manually written in this way. (On the other hand, this reveals the internal more clearly.)

Another important script, `oneStep`, is the top-level ticking script. For each step, what it does is:

1. Call `calcFlameLevel` to hold the sum of flame value at four neighbors.
2. Iterate over the turtles, and if a turtle has non-zero `flameLevel` and its `isBurnt` flag is not set, make it start burning by calling `startBurning` script.

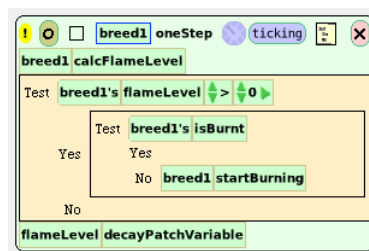


Figure 4.20: The oneStep script for the forest fire simulation.

It again can be constructed in tiles straightforwardly as shown in Figure 4.20.

The other supporting subroutines include:

`setup` Set up the initial values for properties.

`setupTrees` Randomize the initial state of forest.

`setupFire` Set up the initially burning fire.

`startBurning` called from `oneStep`. For the turtles, it sets the `isBurnt` flag, store a positive number (100) into the `flame patch` variable, and store zero into the `wood patch` variable.

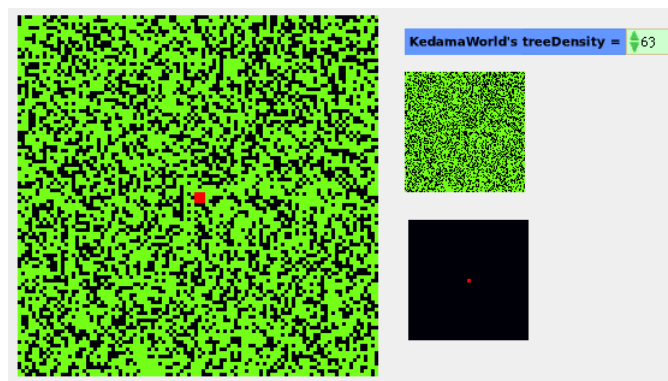


Figure 4.21: A forest initialized with 63% of tree.

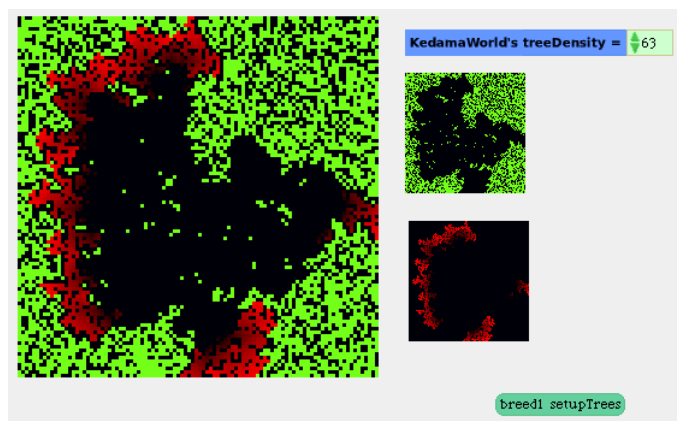


Figure 4.22: A burning forest.

After calling `setup`, `setupTrees`, and `setupFire`, the initial state of the Kedama World and accompanying patch variables will look like Figure 4.21.

Notice that the red area at the center. Then, we start ticking the `oneStep`, the trees will be burnt and the animation looks like a pattern. See Figures 4.22.

The initial density affects how much percentage of trees will be burned in a very non-linear way. There is elaborate analysis in [16] so we omit the detail in this dissertation, but there is a strong threshold around 63%. As long as the initial density is below this threshold, most of trees don't get burnt. However, once the initial density is over the threshold, suddenly *most* trees will be burnt down. This is a very visually revealing example of the non-linear phenomenon in the nature.

This example also has many different explorations: the trees can be made harder to get burnt. There can be some directionality to which the fire spreads better to simulate the effects from wind. Such characteristics can change after some steps to simulate the effects from rain. Trees can grow, and fire sets off at random timing, etc.

On the other hand, these variations don't have significant educational value. The basic model has a lot of interesting phenomenon, but any attempt to make it more *real* would fail, because a cellular automata cannot make a realistic forest simulation. The educators should avoid this pitfall to spend the time on seemingly interesting stuff without the real value.

In regards to the coding style, this example exhibits a work-around that may not be needed if the parallel execution model follows the strict definition of SIMD, where each individual parallel element computes the new state without using the results from other elements. However, this is not the case for Kedama. In Kedama, effects from a turtle is always visible to others. If this example in Kedama was written in a way so that the the new values of patch variables are stored into the patch variables *right away*, the simulation's consistency is broken. Instead, each turtle holds the `flameLevel` property and they used it as temporary memory. Turtles store the new values being calculated into the property, and, at the end of each simulation step, the values in the property of all turtles are written into the `flame` patch variable (logically) at once.

It should be noted that having a temporary variable for each turtle helps factoring the program into smaller sub-steps or subroutines. The user can think the subroutines as input and output in terms of individual turtles. With this observation, we have chosen to go with this approach.

Chapter 5

The Static Aspects of Kedama Language

This chapter presents the details of the language of eToys and Kedama. First, the syntax, the type constraint, and the object model of eToys are explained, as they give the basis of Kedama. Then, they are extended for additions by Kedama. The execution model, or language's dynamic aspects, is discussed in the following Chapter 6.

5.1 The EToys Language

In the following of this section, the syntax of script that the user writes is summarized as the EBNF shown in Figure 5.1. Also, the type constraints in the language are explained. Note that the actual code is written in the GUI tile scripting system that looks like examples shown in previous chapters.

5.1.1 eToys Syntax

The abstract syntax of the language is shown in Extended BNF in Figure 5.1.

In Figure 5.1, nonterminal symbols are represented as capitalized words connected with hyphens in a `monospace` font, and terminal symbols are shown in serif fonts. An entity enclosed by angle brackets is a name defined by the user or the system or an acceptable literal. An asterisk (*), a plus sign (+), and a question sign (?) denote zero or more repetition, one or more repetitions, and zero or one occurrence of the previous element surrounded by curly braces, respectively.

```

Script ::= Header {Top-Level-Stmt}*
Top-Level-Stmt ::= Top-Level-Message
                  | Assignment
                  | Compound-Stmt
Top-Level-Message ::= Object Selector {Exp}?
Assignment ::= Num-Assignment
              | Object-Assignment
Num-Assignment ::= Property Num-Assignment-Op Exp
Object-Assignment ::= Property Object-Assignment-Op Object-Exp
Compound-Stmt ::= Test Test-Exp
                Yes {Top-Level-Stmt}*
                No {Top-Level-Stmt}*
Object ::= Object-Name {Property-Access}*
Property ::= Object-Name {Property-Access}+
Property-Access ::= <property name>
Num-Assignment-Op ::= Object-Assignment-Op
                    | increase by
                    | decrease by
                    | multiply by
Object-Assignment-Op ::= ←
Exp ::= Exp Arith-Op Exp
      | Literal
      | Random Int
      | Property
Arith-Op ::= + | - | * | / | //
           | \\\ | min: | max:
Test-Exp ::= Property Test-Op Exp
           | Property
Test-Op ::= = | ~= | < | <= | > | >=
          | isDisvisibleBy:
Object-Exp ::= Property
            | Literal
Header ::= <script name>
Selector ::= <command or script name>
Object-Name ::= <object name>
Literal ::= <literal>

```

Figure 5.1: The EBNF of the eToys tile scripting language.

For most types, the literal representation (`<literal>`) of a value is available; number literals can be typed into the tile, a color literal is a color box GUI widget, a Boolean is a drop-down box of `true` and `false`, and etc.

`Script` represents a script, and it consists of the header (a script name) and zero or more “top-level statements” (`Top-Level-Stmt`). A top level statement is either a top level message sending (`Top-Level-Message`), an assignment (`Assignment`), or a “Test-Yes-No” statement (`Compound-Stmt`).

An assignment can take two forms. If the property on the left hand side (and the expression on the right) is of `Number` type, it is a numerical assignment (`Num-Assignment`). If these are non-number type, the assignment is called an “object assignment” (`Object-Assignment`). For a numerical assignment, the assignment operator can be either a simple assignment (`Object-Assignment-Op`), or one of three other forms (`increase by`, `decrease by`, `multiply by`) that specify the differential from the current value. For an object assignment, the simple assignment is the only choice (`Object-Assignment-Op`).

A compound statement is a conditional `Test-Yes-No` statement. In the `Yes` and `No` clauses, zero or more repetitions of top-level statements are allowed.

A reference to an object (`Object`) can be an object name (`Object-Name`) or getter call of an object, if the property is in `Player` type.

The right hand side of assignment is an expression (`Exp`) for number type assignment, or an object expression (`Object-Exp`) for a non-number assignment. If the expression is `Number` type, an expression can contain other expressions recursively, a random number generator (`Random`), or literal numbers (`Literal`) combined with the arithmetic operators (`Arith-Op`). For a non-number assignment, the right hand side is either a reference to a property of an object or a property of a property of an object and so on (`Property`), or the literal (`Literal`) that conforms the type of the left-hand side.

The `Test` clause of a conditional statement holds a Boolean-typed expression. The GUI expression editor enforces the constraint, and only the “getters” or property access with no side-effects can be used in the expression.

A selector for a top-level message sending is either the name of a pre-defined command or a user-defined script. Also, every user-defined graphical object has a name.

5.1.2 The Types

The types of eToys system are **Number**, **Color**, **Graphic**, **Boolean**, **Player**, **ScriptName**, **Sound**, and **String**, and some other less common ones. When the user defines a property, type of the property can be later changed from the context menu for the property.

There are type constraints in the language. In summary, these constraints are as follows:

- The left hand side of an assignment must have the same type of the right hand side.
- If the left hand side and right hand side of an assignment are of **Number** type, the number assignment operators are available (i.e., presented in the drop down menu) for the use to choose. Otherwise, only the object assignment is available.
- The arithmetic operators are only available in numerical expressions.
- The expression in the **Test** clause of a conditional statement is **Boolean** type.
- If a property access is used in an object reference, the type of the property has to be **Player**.

Note that the eToys is statically typed; a property can store only the value of specified type. Unlike textually coded languages, the result of type checking is given to the user visually; i.e., when the user drags one tile onto another to combine them, it is accepted only if these tiles satisfy the type restriction.

Some properties are marked as “read-only”. The type-checker of the scripting system prohibits the user to construct an assignment into a read-only property.

However, the type system is not sound; after building a script, the user can change the type of the property later. When the script is executed, the error occurred is caught by the system gracefully and reported to the user, without harming already created user project.

5.2 The Language Model of eToys

EToys uses an object-oriented language model that differs from the underlying Squeak’s. Most notably, the behavior and shape of eToys objects is

instance-based; as written above, the user can modify the slots and methods of an individual existing object. Upon doing this, the individual object changes its shape and protocol immediately. On the other hand, Squeak's objects are class-based, and a change to an object is visible to all instances of the class.

However, the difference is easy to absorb by simulating the instance-based system in Squeak. While the semantics of eToys is not defined in any formal way, explaining the mapping between eToys objects and Squeak objects will help to understand how it works.

In the following, we begin with the explanation of the GUI Framework called Morphic, and then introduce a Squeak class called `Player` that governs the scripting aspect of the eToys, and explain the mapping between them.

eToys is implemented on top of the Morphic GUI framework. In Morphic, a graphical object that the user creates and sees is a (sub-instances of) `Morph`. Morphic provides a good basis for writing a flexible graphic intensive application like eToys. However, Morphic itself is in the Squeak's class-based system; the gap between eToys instance-based model needs to be filled in.

The good news is that one can implement, or simulate, an instance-based system on the top of a class-based one. If the class-based system is flexible enough, the simulation layer can be very thin; for each object, the system may automatically create a class and change the shape and behavior of object by changing the class. This is what the eToys system on Squeak does. Whenever the user creates an eToys object, the system automatically generates a class and makes the eToys object be the instance of the class. Later on, the shape and behavior change of the object is reflected through its class. In eToys, such auto-generated classes are called "uniclasses", as each one of them is supposed to provide the behavior for a unique (or a unique set of) object. The technique itself is, as far as author's knowledge, was used first by ThingLab [44]. As a side note, it is also easy to simulate a class-based system on top of a prototype-based one [45].

In any graphical user interface system, care must be taken not to bind the graphical appearance of an object and its behavior. Even with the uniclass trick, classes are still constrained to the single inheritance hierarchy. Typically, a GUI framework class library provides the class hierarchy of graphical objects (widgets) based on their structural and/or visual similarities [46, 47] and Morphic in Squeak isn't an exception. In other words, the ability to be able to write behavior for a graphical object is a kind of cross-cutting concern [48] for the library that provides the graphical appearances.

The way Squeak solved the problem was to define a class called `Player`

that governs the scripting aspect, and let an instance of `Player` “wear” a graphical object. By using the analogy of an actor on stage and his costume, the `Morph` worn is called “costume”. A `Player` and its costume work together to provide the instance-based graphical object model.

When a graphical object is being added to the scripting environment, a subclass of `Player` and its instance are created, and the instance is made to wear the graphical object. From user’s standpoint, nothing seems to have happened, but a `Player` object is now behind the scene and controlling the graphical object.

The user scripts are compiled into the uniclass of the `Player` instance so that the `Player` can execute it as if it were a normal Squeak code. By mapping scripts to the Squeak method (and compile it with the normal `Compiler`), one can take the advantage of Squeak implementation; in short, the compiled method can be run at the “full speed”.

In a prototype-based object system, “copying” an object can have two distinct semantics. The difference is whether the change of the shape or behavior on one is reflected to others. `EToys` provides both type of copying. Making *siblings* creates a new instance of the same `Player`. The other, simply called *copying* makes a new `Player` class similar to the original player. In this case, the newly created copy behaves same as the original, but the change on one doesn’t get reflected to others.

5.3 The Addition by Kedama

Kedama needs to extend `eToys` language for two reasons. One is the non-intrinsic properties (See Section 2.3.4) that cannot be calculated by the owning object itself but requires extra arguments. The other is newly added data types. To accommodate such objects, the type system needs to be extended. The language extension is small so that the existing users can understand easily. In the following, the syntax addition and type constraint additions are explained.

5.3.1 Kedama Syntax Additions

Figure 5.2 describes the syntax addition to Figure 5.1 by the Kedama system.

The new syntactic addition is `Non-Intrinsic-Property` that represents the non-intrinsic properties as shown in Figure 5.2. An example of a non-intrinsic property is `patchValueIn`. It is treated as a property of a turtle in the language, but its value is not stored in the turtle’s property but a patch cell in the given patch variable. Some properties allows the user to read and

```

Property ::= Object-Name {Property-Access}+
          | Non-Intrinsic-Property
Non-Intrinsic-Property ::= Object Non-Intrinsic-Op Object
Non-Intrinsic-Op ::= 'patchValueIn'
                  | 'upHillIn'
                  | 'angleTo'
                  | 'distanceTo'
                  | 'turtleOf'
                  | 'bounceOn'

```

Figure 5.2: The EBNF of the addition of Kedama.

write, but some others are calculated from other values, therefore cannot be directly stored into (*read-only properties*). The type-checker of the scripting system prohibits the user to construct a statement that would store into such a property.

For example, the `patchValueIn` property is read-write. Therefore, an assignment statement like:

```
aBreed patchValueIn aPatch ← 100
```

is a valid statement (that stores 100 into the patch cells where the turtles in the breed resides), as well as a statement that reads the value:

```
aBreed x ← aBreed patchValueIn aPatch
```

However, a property called `upHillIn` is read-only; therefore, a statement:

```
aBreed heading ← aBreed upHillIn aPatch
```

is valid but not the following:

```
aBreed upHillIn aPatch ← 100
```

(There are some experimental properties that take two or more arguments, but their use is not well-supported.) Other than the fact that these non-intrinsic properties require an extra argument, these can be used anywhere where an ordinary property can be used.

5.3.2 The Type Constraints of Kedama Properties

The syntax additions by Kedama introduce new type constraints. The syntax additions are the non-intrinsic properties that take an extra arguments

and the new type constraint specifies the type of the arguments for such properties.

Table 5.1 shows the constraint for the non-intrinsic properties of turtles. In this table, the first column shows the name of properties, and the second column shows the type of the property (the type of value that the property access returns), and the third column shows the type of arguments for these properties.

Table 5.1: Type constraint for the non-intrinsic properties.

property name	property type	argument's type
patchValueIn	Number	Patch
uphillIn	Number	Patch
angleTo	Number	Turtle
distanceTo	Number	Turtle
turtleOf	Turtle	Turtle
bounceOn	N/A	Turtle

This chapter describes the static nature of the language, i.e., the syntax and the type constraints. However the most interesting part of the definition of Kedama is its parallel execution model. The next chapter is dedicated to discuss the execution model of Kedama.

5.3.3 The Differences between eToys Objects and Kedama Turtles

At a glance, what turtles and their breed can do is similar to what eToys objects can do. For both of them, the user can define/modify scripts and properties, and change the number of similar objects that follows a script. However, Kedama turtles actually have some restrictions.

First, the turtles in a breed have to be *homogeneous* in their shape. The shape of all turtle is exactly the same. The turtles share the same scripts so that when a message is sent to the breed, the implementation of the message (a command or script) is the same for all turtles in the breed. More notably, the notion of copying and creating siblings are changed. The number of turtles in a breed is controlled by a property of breed, and they always share the same shape and behavior. In this sense, all turtles in a breed are similar to siblings, but there is one exception; siblings can have different ticking state for scripts, but turtles obey single status for each script.

Second, the container of the turtles is restricted to a single `KedamaWorld` that is associated with the breed of turtles. All the turtles in the breed are

bound to one Kedama World.

Third, they have a fewer number of built-in commands and properties. Only `forward`, `turn` and `die` are provided as built-in commands. Other commands that eToys objects can respond are omitted. Some of them, such as pen-related ones, can be simulated by using a patch variable. `Show` and `hide` are unified with the `visible` property. Similarly, the set of properties is changed, and generally simplified. Since the turtles are always rendered as pixels on screen, there is no reason to have geometry-related properties. On the other hand, a few non-intrinsic properties are added.

Some restrictions should be lifted (discussed further in Section 10.2), but in general, the removed commands and properties are not relevant (such as drag-and-drop related one) or can be simulated with the other means. The author thinks that the current set of commands and properties provide good basis of writing interesting examples.

Chapter 6

The Parallel Execution Model

The heart of the Kedama system is the parallelism and the emergent behavior introduced by the parallel turtles. Thus, the semantics of the script execution need some consideration.

There is a history of parallel programming languages, but the focus of its usability is on the programmers' and researchers' and they tend to add syntax constructs and directives to specify the type of parallel execution and the assertion on the data to achieve the goal of maximized performance. However, Kedama's goal is slightly different; the parallelism should not only be accommodated in the language, but in a way that is understandable to non-technical users. Performance is of course important, but gaining reasonable (i.e., comparable to a possible implementation in C) performance with minimum learning curve for existing eToys users is more important.

Also, as it will be shown, the flexibility of eToys doesn't allow us to have a perfectly sound semantics model all the time when the user mixes commands for different objects and breed of turtles freely. At the same time, the flexibility should be limited in another way to gain the performance. Therefore, the goal of defining the semantics of Kedama is to provide "good enough" soundness with good flexibility. Let us see how much approximation we can do in this restrictions.

The concern on understandability led us to use a *single thread execution model*; a script execution can be reasoned as if a single thread runs through the script. In other words, even if the data accessed by the statements in a script has some dependency, the end result from executing the script is identical to the single-threaded execution.

On this basic model, there are many possible variations of models. To explore the design space, we have tried three different models. Let us name them “one-by-one”, “simple statement-wise”, and “predicated statement-wise”. These different models stick to the single thread execution model, but have different expressiveness (i.e., the range of possible scripts), different performance, and actually different results. By evaluating these models, the author has chosen the predicated statement-wise model for the current version. In the following, these different models are explained.

6.1 The One-By-One Model

The One-By-One model provides possibly the simplest parallel execution model for turtles. In the One-By-One model, the flow of control can be modeled as this: suppose there is a script that contains a few commands for a breed. When the script is invoked, the first turtle in the breed executes all commands in the script, and then the second turtle does the same, and so on. In other words, the effects from the first turtle’s execution are visible to the later turtles.

To illustrate this model, let us consider this script:

```
script1
  breed1 forward by 1
  breed1 turn by 5
```

When `script1` is invoked, the first turtle in the `breed1` breed moves to its forward direction and then turns. After these two actions are completed, the next turtle in the breed executes the commands, and so on.



Figure 6.1: The execution thread in `script1`.

Figure 6.1 depicts the flow of execution of `script1`. The execution enters from the top of the script, executes the `forward by` action for the first turtle in the `breed1` breed, goes to the `turn by` for the same first turtle, and then move on to the second turtle in the breed, and so on.

The combination of a conditional statement and a script call-out is incorporated straightforwardly in this model. As if each turtle is a full-fledged

object, the script is simply “sent” to the individual turtle.

The problem, however, is its expressiveness. An eToys script allows multiple statements whose receivers are different objects. However, no straightforward semantics can be defined for such scripts in the one-by-one model. For example, suppose there are two breeds `breed1` and `breed2` that have different number of turtles (n and m , respectively), and a script of `breed1` breed as follows:

```
script2
  breed1 forward by 5
  breed2 forward by 5
```

When this script is invoked, what the turtles in `breed2` should do? Some possible solutions would be to move each turtle in `breed2` n times, or only n turtles in `breed2` move once (if m is greater than n); but either one is confusing for the user.

By looking at the static code of `script2`, the “intuitive” semantics would be that all the turtles in `breed1` breed move to the forward direction by 5 once, and all the turtles in `breed2` breed move to the forward direction by 5 once. However, the one-by-one model cannot provide this behavior.

Also, a non-turtle object can cause similar problem. Suppose a patch variable `patch1` is used as follows:

```
script3
  patch1 clear
  breed1 patchValueIn patch1 ← 100
```

This shows a typical sequence of commands that clear the patch variable first and then set the values to the patch cells. The user would expect that the `clear` command is executed just once because it is a non-turtle command. However, following the one-by-one model, it will be executed n times, and only one patch cell will have 100.

One of the solutions for the problem shown in `script2` is to prohibit mixing commands for different objects in a script. However, this restriction would be a big leap from the eToys tile scripting system. It is desirable to have a system where the user can combine commands for different turtles freely, and that does the “right thing” at most of the time, and certainly does not crash when the user does so. Also, it limits the expressiveness. It is typical to want to write a script like `script2`, where some initialization is done first for different objects.

This observation led us further to try different execution models. Also, as discussed later, the implementation of this semantics allocates full-fledged Squeak objects for each turtle, and it exhibits high performance penalty.

6.2 The Simple Statement-Wise Model

One of important observation gained in the previous section 6.1 is that a script should be able to contain commands for different objects. The next attempt, the “simple statement-wise” (SSW) execution model, tries to address this issue along with the potential performance gain.

The characteristics of the SSW model can be summarized as follows:

Single-Threaded It keeps the single-thread execution model. For each invocation of a script, a thread is created and it performs the necessary actions.

Statement by Statement Execution When there is more than one statement in a script, all the actions from a statement is completed before entering the next statement. (In an analogy you can think of the SIMD abstraction, where a statement corresponds to an instruction of a processor and the turtles in a breed correspond to the parallel data.)

One-By-One Execution in A Statement Conceptually, the statement is executed on one turtle by one turtle; if the statement imposes some dependency between the turtles, the output from a turtle is visible to the other turtles.

Sequentialization When a statement doesn’t contain the dependency between the turtles, the system tries to execute the statement “in parallel” on the turtles to gain better performance. If it does, it “sequentializes” the statement and actually executes the statement on individual turtle one at a time. Sequentialization is a semantic preserving de-optimization. The fact that a statement is sequentially executed is not visible to the user (except that the actual performance is degraded).

Conditional Statements Sequentialization From the implementation requirements, a conditional statement is always sequentialized.

Separated Statements Different statements in a script can have different breeds as their parallel breed. Sequentialization occur per statement basis; a script may contain sequentialized statements and parallel statements.

Primary Breed When more than one breed is used in a statement, only one breed is picked as parallel entity, and others are treated as scalar. The parallel entity is called *primary breed*. With the primary breed

concept, a statement can be considered that it is always executed on the turtles in the primary breed.

Let us start with a simple example `script2` again:

```
script2
  breed1 forward by 5
  breed2 forward by 5
```

“Statement-wise execution” means that all the effects from the first command (the command for `breed1`) are completed before the effects from the second statement. In other words, the first command executes the action on the multiple data in `breed1`, and synchronizes for the completion on all of them. Then the control advances to the next command and the command takes action. Since the SSW model keeps the single thread execution model, the flow of execution of `script1` can be depicted as Figure 6.2.

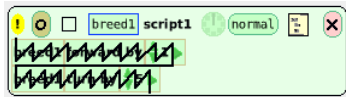


Figure 6.2: The execution thread of `script1` in SSW.

One of the motivations to introduce the SSW model was to give better performance for common cases. While there is still only one thread iterating over the turtles, the basic commands for turtles such as `forward by` and `turn by` can be executed repetitively on the turtles in a breed. This repetition can be optimized by doing it in a C-compiled inner loop.

Let us consider a case where the sequentialization is needed. Suppose we have a script `script4`:

```
script4
  KedamaWorld's sumOfX increase by breed1's x
  breed1's x ← KedamaWorld's sumOfX / breed1's turtleCount
```

Where the `sumOfX` variable of `KedamaWorld` is a scalar property. This is a typical script the user would write to accumulate the `x` values of all the turtles in the breed into the `sumOfX` variable, and then divide the value by the number of turtles to calculate the average. The right-hand side of the second statement is a scalar value, and the assignment stores the scalar value into all the `x` property of the turtles.

The flow of execution of `script4` is depicted as Figure 6.3. Conceptually, this is just the same as Figure 6.2; this means that the model is consistent

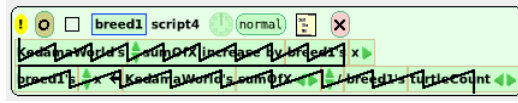


Figure 6.3: The execution thread of script4 in SSW.

thus far. The difference is that the `sumOfX` property is updated just not once but as many times as the number of turtles in the breed. In other words, the output stored in the property is read by the other turtles. Let us call this *data-dependency* in a statement.

The implementation has to take care of such cases. It *sequentializes* the execution of the statement to make the result of assignment by a turtle visible to the next turtle. With this sequentialization, the first turtle fetches the `sumOfX` value, add its `x` and store the partial sum into `sumOfX` property. Then the next turtle does the same with the updated value, and so on. (If the statement is not sequentialized, all the turtles fetches the same value of `sumOfX` first, do the addition and try to store the values into `sumOfX`. In this case, only one turtle's `x` is accumulated in `sumOfX` variable.)

The action by the statements in `script1` as well as ones in `script2` can be executed in parallel for all turtles in the breed. Also, when the second statement is executed, the right-hand side of the assignment produces a scalar value and the value is assigned to the `x` property of all turtles “in parallel”.

A conditional statement is considered to be a large statement, even though it contains many sub-statements and Test expression. The dependency between them have some implication; the execution of the statements in the Yes or No clause for some turtles may modify the result of Test expression for some other turtles.

Also, a conditional statement has a different issue; it requires the ability to execute a command on the selected turtles in a breed. Let us consider a script `script5` as follows. The expected behavior of this script is only the turtles whose `y` coordinates are less than 50 should execute the command.

```
script5
  breed1 forward by 1
  Test: breed1's y < 50
  Yes:
    breed1 turn by 5
  No:
```

How many times, and how, the `breed1 turn by 5` command in the conditional

statement should be executed? Of course, the result of the `Test` expression is different for each turtle, so we have to look at each turtle, evaluate the `Test` expression, and selectively execute the `breed1 turn by 5` statement. How should it be implemented?

In the SSW model, we again have chosen to sequentialize every `Test-Yes-No` statement. The flow of control can be depicted as Figure 6.4. The longer vertical line segment over the `Test` statement means that the `Test` expression for the first turtle is calculated first, and the command in the `Yes` or `No` clause based on the `Test` result is executed next. Then, the control moves on to the second turtle, and so on.

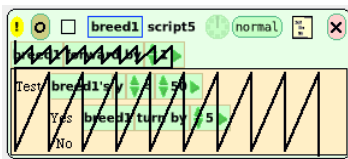


Figure 6.4: The execution thread of `script5` in SSW.

The notion of *primary breed* is needed when a statement contains references to more than one breed of turtle. For example, imagine a script:

```
script6
  breed1's x ← breed2's y
```

Arithmetic and assignment operations for vectors of numbers can have meaningful semantics only when all operands are in the same length or all other operands are scalars. In general, the number of turtles in different breeds, `breed1` and `breed2` in this case, are different. In such case, the meaning of this expression cannot be defined.

A possible solution would have been to check the length of vectors at runtime and raise an error when they don't agree. However, this approach would result in too much runtime errors. In a typical session of eToys, the type constraints provide reasonable static type checking so that scripts are expected to run without runtime error. This property is important to foster the exploratory programming style, as runtime errors are hard to deal with for end-users. Another possible solution would have been to check the breeds in the tiles that the user is trying to combine, and reject the drop gesture when these breeds are not equal. However, we think that this discourages the exploratory programming style. For example, when there is a statement that would look like `breed1's x + breed1's y` and later the user

decided to change it to `breed2'x + breed2'y`. The user may want to substitute the `breed1` occurrence with `breed2` one at a time.

What the Kedama does instead, when more than one breed are used in a statement, is to pick a breed, and designate the breed as the container of the turtles that do the parallel stuff for the statement. Such a breed is called the “primary breed” for the statement. The references to other breeds are treated as the reference to the exemplar. The algorithm to choose the primary breed is simple: the left-most occurrence of turtle in a statement becomes the primary breed. In `script6`, `breed1` is chosen as the primary breed, and `breed2` is treated as the scalar value (read from the exemplar.) In this script, the `y` coordinate of `breed2` exemplar is fetched as a scalar value, and stored into the `x` property of all turtles in `breed1`.

The experience of writing examples suggests that such mixture of references to different breeds in a statement is rarely needed; if some two properties should be always in the same length, they should be properties of the same breed of turtle. If two different breeds should communicate with each other, the user can still use a patch variable to transfer the information. We have decided to choose a breed in a statement as a parallel breed and treating other breeds as scalar, as this prevents runtime errors at small expressiveness cost without restricting the user’s program editing.

The SSW model allows (in most cases) the user to “factor out” a portion of script to create another script; i.e., he can drag-out some statements from an existing script to create a new one, and call the new script from the original site. This is important to allow the user to explore the domain. The user can first create a working (but not cleanly written) set of scripts and later refactor them.

The SSW model works as expected in most of the cases, but it can produce some unexpected results; in other words, this model has semantics discrepancy problem. In below, it is explained by a series of examples.

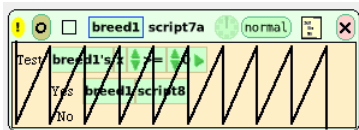


Figure 6.5: The execution thread of `script7a` in SSW.

Suppose we have a script that calls another script that is as follows:

```
script7a
  Test: breed1's x >= 0
```

Yes:
 breed1's script8
 No:

As a conditional statement is always sequentialized, the flow of execution would be as Figure 6.5. Because the test (breed1's $x \geq 0$) is always true for all turtles, script8a is called on all individual turtles in the breed1 one by one.

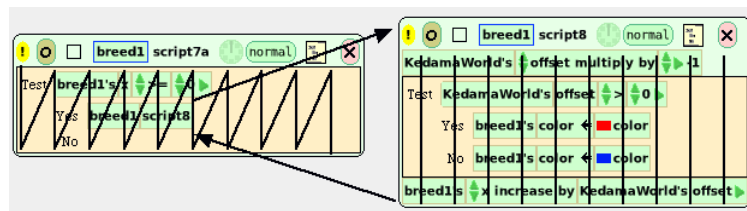


Figure 6.6: The execution thread of script8 when called from script7a in SSW.

Suppose the callee script8 is specified as follows:

```
script8
  KedamaWorld's offset multiply by -1
  Test: (KedamaWorld's offset > 0)
  Yes:
    breed1's color ← Color red
  No:
    breed1's color ← Color blue
  breed1's x increase by KedamaWorld's offset
```

where KedamaWorld's offset is initialized to 1 by the user. If script8 is called from script7a, the flow of execution will look like Figure 6.6. Namely, script8 is sent to individually to each turtle in breed1 because the call site in script7a is already sequentialized. The first statement in script8, is executed n times even though the statement doesn't involve a breed.

However, if the calling script script7b is:

```
script7b
  breed1's script8
```

the call statement in script7b to the script8 is not sequentialized. The flow of execution in this case looks like Figure 6.7. The biggest difference is that the first statement in script8 is executed just once for one script7b invocation. In

such case, all turtles use the same `KedamaWorld's offset` value as the speed of movement. On the other hand, if `script8` is called from `script7a`, half of the turtles move to one way, and rest move to the opposite way. If `script8` is called from `script7b`, all of the turtles move to one way. Such difference in the end-results, caused by an otherwise innocent conditional statement, seems to be an issue on the scalability of the program; when the working-scripts are composed and made so that one script calling others, the user may see unexpected results.

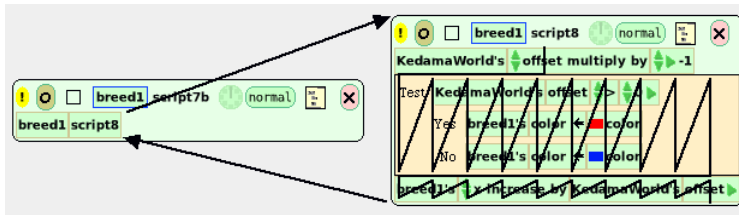


Figure 6.7: The execution thread of `script8` called when called from `script7b` in SSW.

Also note that the sequential execution is much slower than parallel equivalent. For performance sake, it is desirable to stay as much as possible in parallel execution mode.

To overcome the performance and semantics limitations, we have tried a variant of statement wise execution model that we've named "predicated statement-wise".

6.3 The Predicated Statement-Wise Model

The third execution model implemented and tested is called the predicated statement-wise (PSW) model. This model has better characteristics than SSW and is being adapted to the later versions of the Kedama system.

The PSW model inherits much of the characteristics of the SSW model, one large difference is that there is a way to carry the context information (the results from `Test`, for instance) to the enclosed sub-statements.

In the PSW model, a Boolean value that denotes whether the turtle is "alive" or not is attached to each turtle. With predicates, a conditional statement may be executed in the following manner. First, the system evaluates the condition expression in `Test` clause for all turtles. Then, the `Yes` clause is executed in parallel as if there is no enclosing conditional statement; the trick is that only the turtles that have `true` predicate execute the

statements in the clause. The statements in the **No** clause are executed in similar manner but with inverted predicate Booleans.

This idea is very similar to the “masked vector” in High Performance Fortran[33] in the sense that Boolean masks control vector computation execution, and similar to the “predicated instructions” in IA-64 architecture[49] in the sense that the results of computation are not written back to the elements with false predicates.

Let us visit the examples in the previous subsection 6.2. The scripts `script7a`, `script7b` and `script8` were as follows:

```
script7a
  Test: breed1's x >= 0
  Yes:
    breed1's script8
  No:

script7b
  breed1's script8

script8
  KedamaWorld's offset multiply by -1
  Test: (KedamaWorld's offset > 0)
  Yes:
    breed1's color ← Color red
  No:
    breed1's color ← Color blue
  breed1's x increase by KedamaWorld's offset
```

In the PSW model, the results of the **Test** expression are attached to the predicates of each turtle.

When `script8` is called from `script7b`, all turtles enter `script8` in parallel with all of them active, because the default value for the predicate is **true** to make all turtle perform commands. In the case of `script8` is called from `script7a`, the condition expression (`breed's x >= 0`) results in **true** for all turtles as well. Namely, no matter whether `script8` is called from `script7a` or `script7b`, they are consistent and better than the result in SSW.

In `script8`, there is another conditional statement so that it can be seen as nested conditional statements (dynamically) when it is called from `script7a`. Care is taken to handle the predicates in such cases so that the predicates for the inner condition are the logical and of nested conditions.

Again, the results from executing a statement in the PSW model should be identical to the one as if the statement is executed by each turtle one

by one. This means that when the data-dependency exists among turtles, it still needs to be taken care of. The tactics in the PSW model is again to sequentialize such statements.

While the PSW reasonably good semantics and provides enjoyable user-experience, there still remains some semantic discrepancy; when there is a script call-out within a conditional statement and the called script may change the result of `Test` of calling conditional statement. Such condition would have to be sequentialized, but it is not a simple problem.

There are several approaches that could have been taken:

- To perform whole-program analysis and examine the all possible call paths to a script to determine the need for the sequentialization.
- Add syntax for new control structures that explicitly specify the sequential execution and parallel execution.
- To stay on the safe side, and sequentialize all conditionals with any script call out in it.
- Implement a strict SIMD. During the execution of a statement or script, hide the intermediate results from a turtle from others. It involves replicating the data and commits the working copy when the statement is done.

However, Kedama took another much simpler approach in the current implementation; namely, simply to consider any script call-out within a condition statement doesn't affect the `Test` results. The rationale of this approach is as follows.

- Whole-program analysis is prohibitively expensive. Unlike typical programming languages, any script can be an “entry point” (the start node of the call graph of scripts) of program execution. A script may be reached from many different call paths that are the mixture of the parallel mode execution or sequential execution. The system would have to create different versions for a user defined script.
- The introduction of new syntax is cumbersome for the users. It wouldn't allow the user to write a simple script such as `script4` that rely on side-effects. It is more preferable to allow this kind of scripts behave “intuitively” with minimum syntax.
- If we stay on the safe side and sequentialize all statements with script call out, it will lose the performance benefit.

Our experiments have shown that such a case where the called script affects the caller's condition is rare in real examples. (It would be somewhat bad programming style.) The current system seems to provide good enough solution for the users. For future languages evolution, some other approaches are worth to try, especially introducing simple but explicit control structures for parallel or sequential execution.

Chapter 7

Implementation

In this chapter, the implementation of the Kedama system is discussed. The three models described in the previous chapter were implemented and evaluated.

In the following, the three large components in the implementation are explained. The upper layer is the user interface and graphical objects with which that the user interacts. That includes the special `Player` objects for making the instance-based object model. The middle layer is the language processor, that takes the user-defined scripts, transforms it to produce the parallelized version of them. The lower layer is the parallel execution engine that executes the code reasonably efficiently. These are explained in Section 7.1, Section 7.2, and Section 7.3, respectively.

7.1 The Graphical User Interface

One of the goals of Kedama is to be able to give the immediate feedback to the user. A typical simulation that Kedama deals with is dynamic; i.e., not only the result of the simulation, but also the process of the simulation is important. While the user is constructing the simulation, it is often the case that the user would like to change the parameters and scripts while the simulation is running and try the “what-if” simulations without reverting the simulation altogether. In other words, a change on the scripts or variables should be reflected immediately.

Also, the way the eToys system is written, which takes advantage of Squeak’s meta programming ability, is good basis for writing a system like Kedama. For example, what to do when the user requests to add a variable or script is all implemented in Squeak itself. The Kedama implementation

can “hook” these actions easily and insert Kedama specific logic. Also, as long as the Kedama specific user objects follow the same protocol as generic **Morph** protocol, a viewer can be attached to the Kedama specific objects.

The Kedama system defines three user accessible Morph objects. In the following these are explained.

7.1.1 Kedama World

A Kedama World, which represents the “playground” where the parts of the Kedama user program takes place, is implemented as a subclass of **Morph** called **KedamaMorph**. In Figure 3.2, the black square we see at the left is a **KedamaMorph**. **KedamaMorph** is a normal eToys object so that its basic properties such as x, y, etc. are accessible from a viewer. However, its main purpose is a container for the patch variables and turtles.

A Kedama World can selectively display the associated patch variables. That is, a Kedama World can have any numbers of patch variables associated with it and can define a subset of them to be displayed. The same applies to the breeds of turtles associated with a Kedama World. These lists are called “**patchDisplayList**” and “**turtleDisplayList**” and stored in the Kedama World.

For each rendering cycle, a Kedama World executes following steps:

1. Fill the region with the color of itself.
2. For each patch variable in the patch display list, convert the patch cell values into color values with alpha channel and blend the result onto the Kedama World.
3. For each breed of turtle in the turtle display list, check the visible property of turtles in the breed. If it is **true** for a turtle, copy the pixel value stored in **color** property to the grid where the turtle resides. (The alpha-value in the **color** property is ignored in the current implementation.)

A Kedama World holds other instance variables for bookkeeping purpose, but these are not discussed here.

7.1.2 Patch Variable

A patch variable is represented as an instance of **KedamaPatchMorph**. Again, a patch variable is somewhat similar to a grid paper where the user can store values in the cells. The **KedamaPatchMorph** shows the visual representation

of these cells. In Figure 3.2, two squares at the middle are the instances of `KedamaPatchMorph`.

What would be the appropriate data structure to hold the values in a patch variable? A good representation for it is the standard 2D bitmap object in Squeak called `Form`. Because a `Form` holds a 2D array of 32-bit integer data, it is natural to use for the data structure of a patch variable.

Using `Forms` has bonus; one is that Squeak already has a rich set of primitives called `BitBlit` to manipulate `Forms`. Also, during the development of `Kedama`, the data can simply “visualized” by sending (i.e. typing) `display` message, or its variants.

The messages of `KedamaPatchMorph` for accessing (reading or writing) the content of a patch variable are forwarded to the `Form` object associated with it. The messages for bulk mutation (`clear`, `diffusePatchVariable`, and `decayPatchVariable`) are executed by primitives written for `Kedama` that take `Forms` as arguments.

The non-intrinsic properties of turtles require support from a patch variable. For example, to calculate the `upHill` property for turtles, some methods are defined at `KedamaPatchMorph`, because it depends on the content of patch variables.

7.1.3 Kedama Turtle Object

The `KedamaTurtleMorph` is the graphical object with which the user interacts. The interaction includes opening the viewer and writing scripts. It has a few instance variables to support the eToys properties of `Kedama Turtle`, but essentially it is a stub to access the arrays that hold actual data. In Figure 3.2, two small squares at the right are the instances of `KedamaTurtleMorph`.

The “real meat” for turtles is three additional subclasses of `Player` and their instances along with the `KedamaTurtleMorph` instance. As explained in Section 5.2, for each scripted user object, the system creates a subclass of `Player` (uniclass) and its instance to accommodate the instance specific behavior and property. The basic idea is that the system automatically modifies the uniclass class whenever the user makes the shape change or method change of the user object.

For the `Kedama` turtles, it is done by modifying not only one but three uniclasses. Figure 7.1 shows the organization of the classes. When the user create an instance of `KedamaTurtleMorph` (shown as `aTurtleMorph` in the figure) that represents the exemplar, a uniclass called `KedamaExemplarPlayer` (shown as `ExemplarPlayer`) is created under `Player` and its instance is attached

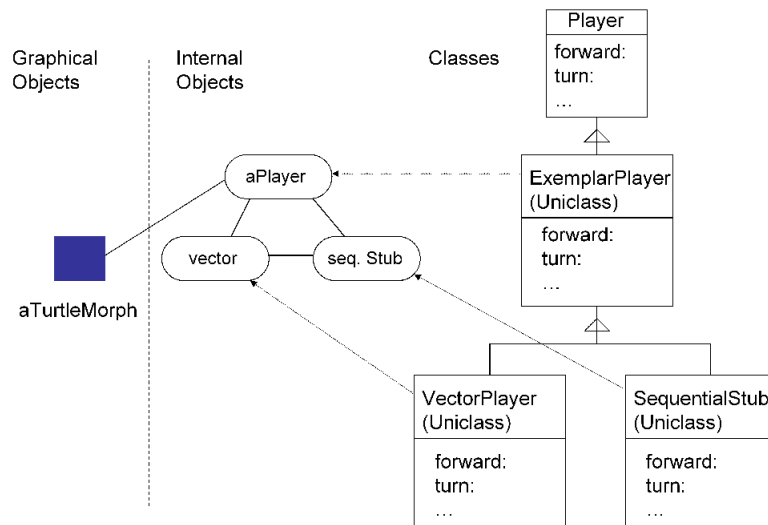


Figure 7.1: The organization of classes for a breed of turtle.

to the `KedamaTurtleMorph` in the same manner a normal uniclass instance is attached to a normal `Morph`.

At the same time, there are two more classes created. One is called `KedamaVectorPlayer` (shown as `VectorPlayer`). An instance of `KedamaVectorPlayer` holds the actual data in arrays (arrays) for all the turtles in the breed, and also the methods to access it. It is made as a subclass of `KedamaExemplarPlayer` in order to keep the external protocol the same. The methods (such as `forward:`, `turn:`, etc.) are overridden to perform vectorized behavior. For example, the implementation of `forward:` at `KedamaExemplarPlayer` makes the associated `KedamaTurtleMorph` on screen move forward by specified amount, while the implementation at `KedamaVectorPlayer` updates the `x`, `y` and `heading` of all the turtles in the breed. Similarly, a getter for a property such as `x` at `KedamaExemplarPlayer` returns the `x` coordinate of the exemplar (in the World coordinate system), while the getter at `KedamaVectorPlayer` returns an array of `x` coordinates (in the associated `Kedama World` coordinate system) of all the turtles in the breed. A user-defined script (script1 in the figure, for instance), is attached to the `KedamaExemplarPlayer` so that the same implementation can be executed by its sub-instances.

Another (and the third) class and its instance created at the same time

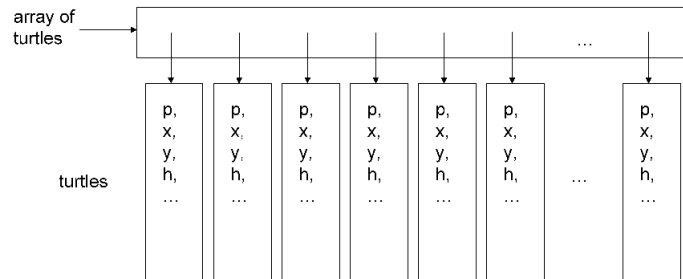


Figure 7.2: The straightforward representation of array of turtles.

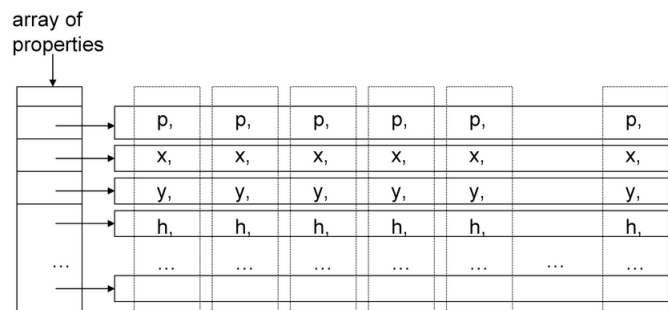


Figure 7.3: The horizontal representation of array of turtles.

is called `KedamaSequentialStub` (shown as `SequentialStub` in Figure 7.1). It provides the “scalar view” on the array of turtles. As explained in Chapter 5, a statement in a script may need to be executed in the sequential mode. Also, when a turtle holds onto a reference to another turtle, the referent behaves as if an individual turtle. For these circumstances, `KedamaSequentialStub` provides the way to specify individual turtle. The messages sent to a `KedamaSequentialStub` cause effects on the specified turtle. For example, the implementation of `forward:` command at this class lets only the specified turtle move forward. Similarly, the getter for the `x` property returns the value of the property of specified object as a scalar value. In short, `KedamaVectorPlayer` and `KedamaSequentialStub` provide the “vector facet” and “scalar facet” of the breed.

The data structure and layout of the data for a vector of turtles is somewhat unusual. A straightforward data layout that was used for the One-By-One model (Section 6.1) is to allocate a full-fledged object for each turtle; in other words, to split data “vertically” so that the properties for a turtle such as `x`, `y`, `heading`, and subsequent properties are placed continuously in physical memory (See Figure 7.2). However, the implementation of the statement-wise models, the data is cut “horizontally” (See Figure 7.3); namely, the same property for all turtles is held in one array and placed continuously in physical memory. The floating point numerical data is represented as IEEE 754 32-bit floating point and placed continuously so that they are accessible as “`float[]`” from the C-compiled primitives. Such an array is an instance of `KedamaFloatArray` in Squeak. In this way, both C-compiled code and Squeak level code can access (read and write) slots of the array to achieve both the performance of C and flexibility of Squeak code. Similarly, color values are stored in a `WordArray` as 32-bit values so that primitives can access them as “`unsigned int[]`”. The properties in Boolean are represented as `ByteArray` so that primitive can access them as “`char[]`”. When the number of turtles is changed (via the `turtleCount` property), the arrays that represent the intrinsic properties and user-added properties are resized (i.e., newly allocated) to the new `turtleCount` value.

When a property is added to a breed of turtle, an appropriate type of array is created and attached to the breed. Along with the array, three pairs (getter and setter) of accessors for the property are generated. One pair is for the exemplar object to access the exemplar object’s property. The second pair is created at `KedamaVectorPlayer`; the accessors access the property as vector; the getter for `x` returns the entire `KedamaFloatArray` for `x` property. The setter for `x` takes an argument either a scalar value or a `KedamaFloatArray` and store the scalar value into all slot of array, or copy

the value slot by slot. If the property is in `Color` type or `Boolean` type, and a scalar value (an instance of `Color` or `Boolean`) is passed to the setter, the value is converted to 32-bit or 8-bit integral value and stored to all the slots. The third pair is for `KedamaSequentialStub`; the getter returns the property of the specified turtle (i.e., the slot at the specified index of the vector for the property) as a scalar value. The setter gets a scalar value and store into the array at the appropriate slot. The static type system of the eToys and Kedama system ensure that there is no case where a scalar setter gets an array as argument.

This horizontal representation also reduces the garbage collection overhead tremendously. In the vertical representation, an array of turtles holds on to the pointers to objects that represent turtles. Since the garbage collector traverses all pointers in the marking phase, this is heavy overhead especially if the number of turtles is more than several thousands. On the other hand, in the horizontal representation, the long arrays are homogeneous data arrays that are known to be non-pointers. The garbage collector skips the traversal from such arrays.

The downside of this horizontal representation is that the turtles in a breed have to be really *uniform*. The shape of all turtle is exactly the same, and when a message is sent to the breed, the implementation of the message (a command or script) is the same for all turtles. Some full-fledged object-oriented SIMD computation models allow the objects in a collection to be of different classes and provide different implementation of the same method; this is not the case for Kedama.

When a command such as `forward` by is executed on a breed of turtle, the message is first sent to the `Player` (`KedamaExemplarPlayer`) and then “delegated” to the associated `KedamaVectorPlayer`. The version of `forward` by at `KedamaVectorPlayer` uses `KedamaFloatArrays` that represent `x`, `y` and heading property, and some other data and modifies the arrays. When the command is executed in the sequential mode, a `KedamaSequentialStub` is used and the action is performed only on the specified turtle,

An instance of `KedamaSequentialStub` specifies the associated turtle by its `who` property. To execute an action or to access the property of a particular turtle, the `who` value is set to the `KedamaSequentialStub` instance itself. Then, the actual index into the arrays held by `KedamaVectorPlayer` is calculated from the `who` value, and proper elements in the arrays are accessed. The reason not to hold onto the actual index in the first place is that turtles may die during the execution; the elements at the slots for the dead turtles in the arrays are removed and the arrays are compacted. Because of the compaction, the index for a particular turtle may move.

The calculation from `who` to actual index may seem to be expensive operation, as it requires the searching for an element in an array. However, the `TurtleVectorPlayer` keeps cache of the mapping that is flushed only when a turtle died.

7.2 The Language Processor

The language processor of the Kedama system takes the parse tree constructed from a user-defined script, analyzes it and transforms it, and produces a `CompiledMethod` that properly utilizes the primitives in the parallel execution engine.

A sidenote: for the One-By-One model, the transformation is virtually unnecessary. The given script is directly compiled into an (almost) equivalent `CompiledMethod`. The `CompiledMethod` will be invoked on each turtle (a full-fledged object) one by one when the script is executed. (end of sidenote)

The stages to generate a `CompiledMethod` from the graphical tiles are as follows:

1. Generate a string representation from the graphical tiles.
2. From the string representation, create the parse tree.
3. Attach “attributes” to nodes in the parse tree and calculate them in a manner of Attribute Grammar.
4. Transform the parse tree based on the attribute values.
5. Generate the `CompiledMethod` from the transformed parse tree.

In the following, these stages are explained. Obviously, the item 3 and 4 are the most important ones and explained in detail.

7.2.1 Generating the String representation of a Scriptor

A script editor (“`Scriptor`”, in short) is a graphical object that acts as a container for other graphical tiles that represent the program. The user can drag and drop the tiles, or sub-instances of `TileMorph`, to construct a script. The `TileMorph` class has 16 subclasses, each of which represents slightly different constituent in the language, such as an assignment, a random number generator, literal values in different types, and etc. These tiles hold the information that the tile is representing.

The subclasses of `TileMorph` implement a polymorphic method called `storeCodeOn:`. Upon the compilation of a script, the `storeCodeOn:` message is recursively sent from the root of scriptor and all the submorphs put its textual representation into the text stream passed as the argument of the message. At the end of the recursive call, the stream stores the string representation of the script in Smalltalk syntax.

7.2.2 Creating Parse Tree

The string representation created above is compiled into tree-structured sub-instances of `ParseNode` by the `Compiler`. One may think that the class hierarchy under `ParseNode` should be relatively simple, because there is virtually only one construct (message sending) in the language. While there are about 20 classes that represent different node in a parse tree, only about 4 of these have significant meaning to the parse tree transformer explained in Section 7.2.4. The leaves such as literals, variables, selectors, etc. are represented as distinct classes. For making recursive structure, there are classes for a message send (`MessageNode`) and a block construct (`BlockNode`). These two recursively contains other sub-instances of `ParseNode`. A class called `MethodNode` is used to represent the top-level, or “start symbol” in the BNF sense, for the tree that represents a script.

7.2.3 Attribute Evaluation and Evaluator Generator

The core technique used in the language processor is drawing upon the idea of “Attribute Grammar” (AG) [50, 51]. An attribute grammar is a declarative specification of syntax and semantics. The syntax specification is given as a form of BNF, and the semantics specification is given as a set of functions to calculate “attributes” (named values) attached to the nodes in a tree. The value of an attribute is defined in terms of a function (“attribute function” or “attribution rule”) whose inputs are other attributes. To accommodate the complexity of recursive data structure (i.e., a tree), only the “local” attributes, or attributes of nodes directly connected to a node can be used as inputs of an attribute at the node. Also, the single assignment property, where each attribute is evaluated just once, contributes the clean definition of AG.

Typically, a specification of AG is read by a software system called “attribute evaluator generator” and it generates an “evaluator” engine that accepts a tree and calculate attributes for the tree. On the other hand, an attribute evaluator can be written by hand. In either case, such an evalu-

ator recursively traverses the given tree in the appropriate order based on the tree's shape and calculates the attribute values in proper order. Generating an evaluator is usually preferable, because a clean specification can be analyzed and given mathematical proof of its property. On the other hand, writing the evaluator by hand is a viable approach especially when the syntax is fixed, and the attribution rules are relatively simple. In the Kedama's case, the parse tree that the attribute evaluator deals with is fixed to be a parse tree constructed for a method in Squeak. Also, the attribution rules for implementing the SSW model are relatively simple (compared to PSW); they basically look for conditional statements or the number assignment operators that take difference from the current value (as they introduce the data-dependency), and sequentialize them. It was conceivable to write the evaluator for it by hand.

Based on this observation, the attribute evaluator for the SSW model was written by hand. The evaluator visits all nodes in the given tree once in the depth-first manner to calculate the primary breed of each statement, and visits again to determine the presence of data-dependency in depth-first manner. It uses 6 attributes, and a several (large) hand-written recursive Smalltalk methods for traversing the tree. It was an error prone task, but it worked well.

After the implementation of SSW, the author realized that the implementation of the PSW model requires more complicated attribute evaluation. The conditional statements needs to be transformed differently based on the results of dependency-analysis, and the analysis requires passing the information between the Test expression and the body. This is too complicated to write by hand. The author decided to write an attribute evaluator generator and the input for the generator (the specification of attribution rules) for the PSW model. Let us call this evaluator generator and driver of the evaluator `KedamaEvaluator`.

An attribute grammar defines the grammar's syntax in BNF and the semantic rules but `KedamaEvaluator` primarily deals with the Smalltalk syntax tree that the standard Squeak Compiler produces; i.e., the syntax is not defined in the grammar but just use the Smalltalk's class definitions as its input.

`KedamaEvaluator`'s biggest departure from the conventional attribute evaluators is that it deals with a flat list (of sub-trees). When a syntax node such as one that represents a message sending, the arguments nodes can be a flat collection with zero or more elements. To select the appropriate semantic rule for a particular shape of trees, the rule selection is done dynamically.

The class of the grammar for `KedamaEvaluator` can be considered as a non-circular AG [52]; i.e., there shouldn't be any cycle in evaluation-time dependency among the attribute occurrences. When the evaluator takes a Squeak parse tree, it instantiates and attaches the attributes to all nodes in the tree. Then, based on the dependency specified in the specification, it topologically sorts the attribute instances on the tree. Finally, `KedamaEvaluator` applies the appropriate attribute function for each attribute in the sorted list.

The AG specification for `KedamaEvaluator` is object-oriented, in a sense that there is a class hierarchy among the parse tree nodes. The specification allows us to define different attributes for different subclasses of `ParseNode`. (It certainly doesn't have, or not intended to have, the formalism provided by the Object-Oriented AG [53].) To select the attribution rule for an attribute at a node, the most specific attribution rule is selected dynamically based on the inheritance chain and the tree shape around the node.

The AG specification allows higher-order attributes as well [54]. In the attribute functions for `KedamaEvaluator`, the node and its sub nodes are accessible as objects so that they can be used in the calculation of resulting attribute. By allowing the higher-order attributes, the tree transformation in the subsequent stage becomes simpler. While it is possible to make one attribute depends on a remote attribute through a higher-order attribute, the topological sorter cannot handle such dependency. It is the grammar-writer's responsibility to ensure that the value is calculated before. However, the specification of PSW doesn't use such remote attributes.

In the following, the actual specification for the PSW model is explained. In a script, a reference to an object (a `Player` for majority cases) is usually "direct"; namely, a `Player` in a script is directly and unambiguously referred to by its name or the pseudo-variable "self". (The exception is when it is stored into a property of an object.) The language processor relies on this fact to statically determine the "types" of objects in a script.

There are about 23 attributes and 59 rules for the PSW specification. Most of the attributes are internally used to calculate others, but a few are carried to the next tree transformation stage. These attributes and the transformation are explained in appendix C.

7.2.4 Parse Tree Transformer

After calculating all attribute values on a parse tree, the language processor transforms the parse tree in a manner of attribute-value directed transformation.

The purpose of the transformation is to insert `doCommand:` or `doSequentially:` message into the statements that involve a breed of turtle. These messages specify whether the statement is executed in the parallel mode or sequential mode.

For example, suppose we have a script owned by a breed called “breed1”:

```
script10
  breed1 forward by 1.
```

it will be transformed into:

```
script10
  self doCommand: [:t1 | t1 forward: 1].
```

Where, The square brackets ([and]) specify a *block* that is analogous to a closure in functional languages. The colon and a name (`t1` in this example) is an argument to the block. The `doCommand:` message passes an appropriate object to the argument of the given block and evaluates the block. The `self` pseudo-variable may refer to an instance of one of the three unclasses; when the script is executed as an entry point of execution, `self` will be bound to an instance of `KedamaExemplarPlayer`. Upon the execution, the block is delegated to the associated `KedamaVectorPlayer` and subsequently, the `KedamaVectorPlayer` executes `forward:` message. As a result all turtles in the breed move forward by 1.

One may think that whether sequentialization is needed can be decided statically. If it were the case, there is no reason to put the “indirection”, or an extra message sending”, for each statement. However, depending on the call path to the script, the receiver of the script may change dynamically and the same statement should behave differently. That is why this indirection is placed to absorb the difference.

Table 7.1: The action for the `doCommand:` and `doSequentially:` message for different receivers.

	<code>doCommand:</code>	<code>doSequentially:</code>
<code>KedamaExemplarPlayer</code>	delegate	delegate
<code>KedamaVectorPlayer</code>	evaluate	repeat
<code>KedamaSequentialStub</code>	evaluate	evaluate

Table 7.1 summarizes the actions. In the table, “delegate” means to pass the same message to the associated `KedamaVectorPlayer`, “evaluate” means

to evaluate the given block with itself, and “repeat” means to have the block iterate over the turtles by using a `KedamaSequentialStub`.

For example, suppose we have these two translated scripts `script11` and `script12`. In this example, `script12` is called from `script11`.

```
script11
  self doSequentially: [:t1 | t1 script12]

script12
  self doSequentially: [:t1 | t1 forward: 1]
```

If the user invokes `script11`, the `KedamaExemplarPlayer` will be bound to `script11`'s `self`. In this case, the `doSequentially:` message is delegated to the associated `KedamaVectorPlayer`, and on the `KedamaVectorPlayer`, it iterates over the individual turtles by passing `KedamaSequentialStub` to the given block (i.e., `t1` will be bound to the `KedamaSequentialStub` object).

Conditional statements are transformed differently in the SSW and PSW models. In the SSW, a conditional statement is not an exception and just enclosed in a `doSequentially:` block. In PSW, however, a conditional statement is treated differently from other statements. Suppose we have a conditional statement in a script: `statement`:

```
script13
  Test: (breed1's x > 50)
  Yes:
    breed1's turn by: 20
  No:
    breed1's turn by: -20
```

It will be transformed into:

```
script13
  self doCommand: [:t1 |
    t1 test: (breed1 x > 50)
    ifTrue:
      [:t1 | t1 doCommand: [:t2 | t2 turnBy: 20]]
    ifFalse:
      [:t3 | t3 doCommand: [:t4 | t4 turnBy: -20]]]
```

Namely, a message node that represents a conditional statement, whose selector is `ifTrue:ifFalse:`, is transformed to a node with `test:ifTrue:ifFalse:`. The first argument for the message is the result from the condition expression (`breed1's x > 50`, in this example), and the second and third arguments are the **Yes** and **No** clause with receivers replaced. The result of the condition expression is an array of Boolean. The array is passed to the `breed` object that is bound to `t1`, and the slots in the array are attached to each turtle as predicates. Then, the **Yes** clause and **No** clause are executed with the predicates.

While doing this transformation, the data dependency among turtles is checked. If the entire statement should be sequentialized, a message named `doSequentially:` that specifies the sequential execution of given block is used as follows:

```
script14
  self doSequentially: [:t1 | t1 test: (breed1 x > KedamaWorld offset)
    self ifTrue:
      [:t1 | t1 doSequentially: [:t2 | t2 turnBy: 20.
        KedamaWorld offset: KedamaWorld offset * -1.0]]
```

More formal transformation rule is described in the Appendix C.

7.2.5 CompiledMethod Generation

After transforming the parse tree, a `CompiledMethod` for the parse tree is generated. To calculate the temporary variable index for newly introduced variables by the transformation, the transformed parse tree is decompiled to a textual representation once, and compiled normally.

7.3 The Execution Engine

The execution engine consists of a set of “primitives”. A primitive is a C-compiled routine that is called from Squeak code, and typically provides faster execution of some intensive computation, or access some OS API. These primitives can be grouped together in shared library files, and such a library can be loaded to the Squeak VM on demand. If a primitive doesn't need any platform dependent features and its purpose is solely for optimization, a primitive can be written and debugged in the Squeak's interactive environment, and later translated to ANSI-C compliant code and compiled

by an optimizing C compiler. All primitives for Kedama are platform independent so that they are developed entirely in Squeak.

As written in Section 7.1.3, the data for the turtles in a breed is sliced horizontally and stored in several homogeneous arrays in Squeak. Such arrays has the same bit and byte layout as the C language arrays, so a primitive can iterate over the arrays to perform certain operations at “the speed of C” (i.e., a function compiled by an optimizing C compiler). Some of the primitives take a predicates (a `ByteArray`), and perform the operation only on the indices specified by the predicates.

The primitives can be grouped into 1) arithmetic operations, 2) logical operations, 3) predicated array assignments, 4) turtle actions, and 5) others. In the following, the primitives are explained by this grouping.

More details on the primitives are described in Appendix D, but the summary is given here.

Let us take the primitive for addition, one of the arithmetic operations, as an example. This primitive takes two arguments that may be two arrays of numbers or an array of numbers and a scalar. For the array-array case, the values at the same index from these two arrays are added and the results are stored into the third (result) array. For the array-scalar case, the scalar value is added to each value in the argument array and the result is stored into the result array.

Care is taken so that different types of arrays and scalars can be mixed; namely, `KedamaFloatArray` and `WordArray` can be used for vector arguments, and `Float` and `SmallInteger` can be used for a scalar argument, and they produce correct results.

A logical comparison operation is similar but the results (Boolean values) are stored into a `ByteArray`.

A predicated array assignment primitive assigns a value or values to the destination array but only at the slots specified by the predicates. Its arguments are a source value in a scalar or a vector and a `ByteArray` that represents the predicates. The `ByteArray` is treated as a vector of Boolean, and only slots in the destination vector that have true in the corresponding slots in the `ByteArray` get updated.

The turtle operation primitives are specialized for the basic actions for turtles. The commands such as `forward by`, `turn by`, etc. have the implementation in the primitives. When a `KedamaSequentialStub` is executing a command or an assignment, there is computation that involves substantial floating point computation. For example, `setHeading` for one slot involves `sin`, `cos`, the normalization of the resulting angle, etc. The problem in Squeak is that Squeak allocates real `Float` objects for all the intermediate results if it

is done at the Squeak level. To optimize these operations, there are versions of primitives that are used in the sequentialized commands.

The “other” primitives include the supporting primitives such as storing random numbers into an array and magnifying a bitmap by an integral scale factor, etc. With the primitives for turtles and vector operations, other methods with loops become the major obstacles to the performance. The primitives in this “other” category contribute to reduce the bottleneck at these areas.

Chapter 8

Evaluation and Discussion

This chapter presents an evaluation of the system and discusses future directions.

8.1 Expressiveness of Language

Not unlike other parallel programming languages, there are two aspects to the expressiveness discussion of the language of Kedama. One aspect is the ability to specify the parallelism, and the other is the selection of built-in primitives for commands and operators. Also, the semantics of the language should be designed to provide a good basis to write complex programs. These issues are discussed in the following paragraphs.

The Syntax For Parallel Execution There is no special syntax to specify the parallelism. A Kedama user usually has some experience in eToys, and understands the basic model of eToys; where an object does something when a message is sent. It is very similar in Kedama with one difference. In Kedama, every object in the breed does the same thing. In other words, the user just thinks about a turtle, relates himself to the turtle, and writes commands for it. Then, the other turtles along with it do the same action. This model works perfectly if the action performed by a turtle doesn't affect the other turtles in the breed.

Soundness of Semantics While the parallel turtle abstraction works reasonably well for an end-user's need, there are areas where some explicit parallel construct might help. Also, as described in Chapter 6, the system doesn't provide completely clean semantics in the area of how side-effects

are handled, even though the sequentialization cures the issue in *most* of the cases. Also, a programming language should let the user write small and meaningful units of work (functions, methods, or scripts) and compose them (i.e., one calls another) in an orthogonal manner. As described in the forest fire example (Section 4.3), a simulation can be written in this way.

There are a few possible (and known) ways to overcome the parallel semantics problem, but the current design of Kedama is toward performance and a smooth learning curve.

One of the potentially viable solutions to achieve better semantics is to add explicit control structures that represent blocks to specify a sequential or parallel execution. An educational programming system, Alice [30], has such explicit control structures. While this explicitly specifies the execution mode, this is cumbersome at the same time. Experiments show that the need to be able to write a simple script, such as `script4` in Chapter 6, has more importance than the ability to specify parallelism explicitly. For an extension of Kedama where the aim is to provide a more general-purpose language, this may be a feature to try.

A whole-program analysis could find the parallelism better, as data-dependency across the script boundary can be taken into account. If the system could figure out all incoming calls path to a given script, we could avoid making the decision whether the statements in the script should be sequentialized or not. Instead, we could make and keep variations of the same script internally, and execute one of them selectively. Unfortunately, this is not practical because the number of versions for a script can be exponential to the number of written properties in the script. If the program has one or a very limited number of entry points, the calling path would be limited. However, in Kedama and eToys, every script can be invoked by the user and therefore can be the entry point of a script.

Another viable approach is to introduce strict SIMD semantics. In this model, during the execution of a statement or script, the intermediate results from a turtle is hidden from others so that the each turtle can have a view of data without caring about the “interference” from other turtles. This involves copying the state of the system and letting the turtles read the data always from the old state. This is doable but adds some overhead. More importantly, this does require the explicit control structures in the syntax. However, again, this model would be more adequate for a more advanced environment. Among other things, this semantics would give a good answer to a question that a child might ask: “why can’t this turtle read the value written by that turtle here?”.

The parallel execution model that various versions of StarLogo (except

the first one on the Connection Machine) and NetLogo take was the multi-threaded model where side-effects are visible to other turtles. In this model, an invocation of a script allocates a number of threads for the turtles. This model is “intuitive” in the sense that it is closer to treating the turtles as independent and autonomous entities. (In the documents written by the implementers of StarLogo [19] and in personal communications with them, the illusion of independent and parallel motion of turtles is stressed.)

From the one-by-one model, Kedama could have taken the path of the multi-threaded model; however, Kedama took the path for the statement-wise models for performance reasons. The multi-threaded model tends to be slow when there are a few thousands turtles in the system. The real problem is that the performance is not linear to the number of turtles in the multi-threaded model. In fact, NetLogo and StarLogo freeze when the user allocates more than 200,000 turtles or so. This is not enough to write a pixel manipulation program whose world size is comparable with the screen size. While the multi-threaded model can be optimized by doing careful tuning, including the hardware stack manipulation [55][56], doing so in cross-platform and portable way is not practical.

8.1.1 The Language Features

Limitations For Gaining Performance From another viewpoint, one of the biggest limitations of Kedama’s expressiveness is that it doesn’t have a way to specify the neighboring turtles, nor to read the value of the neighboring patch cell, as there is no mechanism for indexing a turtle in a breed nor a patch cell in a patch variable. The interaction is usually written via a patch variable even though it is possible to describe the direct interaction between turtles. The typical and suggested style is to describe the interaction between turtles via a patch variable. This prevents the user from writing numerical parallel programs, such as matrix factoring and finite element method simulations, where indexing the elements is essential. This, of course, will be important for widening the audience and an interesting aspect toward generalizing the language.

One more thing to note is that a breed is limited on the same behavior on a uniform array of turtles. This simplifies the system and gives a way to implement a system with C-comparable performance. At the same time, one would imagine an array with objects of different classes. The statement-wise execution model would require the synchronization at each statement. This will add more flexibility to the language, but the author thinks that this is overkill for the classrooms’ use.

The Arithmetic Operations The arithmetic operations in Kedama are limited for a few reasons. The limitation comes from a few factors; a) for the sake of a smooth transition for eToys users, b) the philosophy of the design of an environment where there are not too many “black boxes”, and c) merely the implementation limitation of eToys.

EToys supports basic arithmetic such as addition and multiplication, etc. However, operations “one-level above”, such as the square root function and trigonometric functions are omitted. One of the reasons for this is because these operations can be implemented by the users themselves. For example, the square root function can be written in the Newton-Raphson method and the trigonometric functions such as the sine function can be written using the **turn by** and **forward by** primitives. In a sense, a sine function should be written and understood by the user, rather than just given as something to use.

The eToys’ philosophy was to give the students the motivation to understand the function by writing it by themselves. This design decision has worked well for the eToys’ target age group, namely for 11 to 12 year olds. They rarely needed these functions in their projects.

On the other hand, it also turned out that older students have a different reaction when they found they missed such functions. Once their projects involve something as simple as measuring distance between two objects, the square root function becomes very useful.

Of course, there is always an argument whether an educational system should or should not provide a feature or a function just because it is useful. If the user doesn’t understand a feature, providing it as a built-in function would not help the user to learn it. On the other hand, if the user understands the feature, having to implement it in every project is cumbersome and may prevent the user from learning new things.

In addition, Kedama has chosen to provide some of the quadratic functions. The assumption is that a typical user has some knowledge in such functions already. For example, a turtle in Kedama provides non-intrinsic properties called **distance to** that measures the distance to a given turtle, **angle to** that gives the heading angle to given turtle. (Note that Kedama doesn’t provide it as the generic square root function; partly because the demand is rare, and partly because the tile-scripting system of eToys was not designed to accommodate such unary functions.)

	# of turtles	script	result (ms)
(1)	10	simpleTurn	5.5
(2)	1,000	simpleTurn	633.8
(3)	100,000	simpleTurn	73272.0
(4)	10	condTurn	7.0
(5)	1,000	condTurn	647.3
(6)	100,000	condTurn	75046.5

Table 8.1: The results from the One-By-One execution model.

	# of turtles	script	result (ms)
(7)	10	simpleTurn	0.4
(8)	1,000	simpleTurn	14.2
(9)	100,000	simpleTurn	1456.4
(10)	10	condTurn	4.8
(11)	1,000	condTurn	338.4
(12)	100,000	condTurn	32535.0

Table 8.2: The results from the SSW execution model.

	# of turtles	script	result (ms)
(13)	10	simpleTurn	0.4
(14)	1,000	simpleTurn	14.4
(15)	100,000	simpleTurn	1476.8
(16)	10	condTurn	1.2
(17)	1,000	condTurn	14.4
(18)	100,000	condTurn	1410.2

Table 8.3: The results from the PSW execution model.

	# of turtles	script	result (ms)
(19)	10	simpleTurn	0.5
(20)	1,000	simpleTurn	55.2
(21)	100,000	simpleTurn	301,800
(22)	10	condTurn	0.6
(23)	1,000	condTurn	91.0
(24)	100,000	condTurn	310,100

Table 8.4: The results from equivalent code in NetLogo.

The result columns show the elapsed time in milliseconds for 100 repetitions measured on an Intel Pentium-M 1.2GHz computer.

8.2 Performance Evaluation

The purpose of performance evaluation here is to compare different execution models and different systems. We begin with measuring the results from micro-benchmarks to see how long a simple operation takes to execute, and macro-benchmarks to get the sense of the practical improvements from the micro-level improvements. The interesting points to investigate are: at micro-level, 1) the overhead for checking the predicate vector in primitives and 2) the performance difference when conditional statements are involved, and at macro-level, the visible performance improvement in typical examples.

Table 8.1, 8.2, and 8.3 show the results from the One-by-One, SSW, PSW execution models, respectively. Also, Table 8.4 shows the results from the equivalent code written in NetLogo. A one-liner script `simpleTurn` contains only one `turn` by command. The script `condTurn` is similar, but the same `turn` by command is placed in a conditional statement, whose test looks like “turtle1’s $x \geq 0$ ”, and therefore always returns “true” for any turtle. The scripts are repetitively executed 100 times with a different number of turtles. The running time (average of 4 samples) has been measured on a computer with a 1.2GHz Pentium-M processor and shown in milliseconds.

The first thing to notice is that `simpleTurn` in the One-by-One execution model is one order of magnitude slower than the other two, because there is no support from primitives in the execution engine. By comparing the results of `simpleTurn` from the SSW and PSW model, it can be said that the overhead of the predicate vector check is very small. By looking at the results from (9) and (15), the overhead is only about 1.4%.

For a conditional statement, the SSW model loses support from the primitives as it has to be sequentialized. As a consequence, the result becomes only twice as fast as One-by-One. The PSW model gets similar results from `simpleTurn` and `condTurn`, as both are executed in the parallel mode. As for `condTurn`, by comparing the results from (12) and (18), or (11) and (17), one can tell that avoiding sequentialization gives us a factor of 23 improvement at the micro-level over the SSW model.

Also, it is worth noting that the statement-wise models, especially the PSW, are reasonably fast in terms of its absolute speed. For example, the result (15) shows that turning 100,000 turtles 100 times takes 1476.8 milliseconds on a 1.2 GHz Pentium-M computer. This means that one turn operation takes 177 clock cycles¹. Given that the system is dynamically

¹Note that Pentium-M’s instruction per clock is more than one.

	Example names and # of turtles		
	LifeGame 10,000	ForestFire 10,000	GasTank 2,000
NetLogo	3.6	3.9	14.7
NetLogo (opt.)	27.8	27.5	-
SSW	9.0	20.0	93.0
PSW	39.0	45.0	79.0 (102.0 ^(*))

Table 8.5: The results from macro-benchmarks in different execution models and systems.

The results show the numbers of frames per second (fps) from the real-time animation on an Intel Pentium-M 1.2GHz computer.

modifiable and involves intensive floating point number calculation, it can be said that this is a reasonably good performance.

Let us now compare the numbers between PSW and NetLogo. As you can see, when the number of turtles is 10, the numbers are somewhat comparable, and even with 1,000 turtles, the numbers are within an order of magnitude range (14.4 vs. 55.2). However, if the number of turtles is 100,000 it is simply incomparable. This discrepancy comes from the fact that the performance on NetLogo, which employs a multi-threaded execution model, doesn't exhibit "linear" scalability; that is, the execution time is not proportional to the number of turtles. On the other hand, Kedama's performance is linear to the number of turtles, so that its execution time is predictable with large number of turtles.

Here, we take a few real examples and compare the performance with different execution models in Kedama and NetLogo, the fastest preceding work.

Table 8.5 summarizes the measurements of performance from different examples on different execution models and systems. The results show the numbers in frame per second; i.e., bigger numbers are better. Showing a frame involves the all calculation, rendering of the results into bitmaps, and sending the bitmaps to the screen. It is a good measurement for assessing the real value of micro-level improvements.

The first column shows the different implementations. The "NetLogo" row shows the results from straightforward implementations of the models. The "NetLogo (opt.)" row shows the numbers from optimized implementations where the code is modified so that the number of turtles in the system is limited. The "SSW" row shows the numbers from the implementation of

SSW model. The “PSW” row shows the numbers from the implementation of SSW model. For “SSW” and “PSW”, the examples are written straightforwardly, or in equivalent form of “NetLogo” implementations. There are three examples; LifeGame, ForestFire, and GasTank. ForestFire and GasTank are explained in Chapter 4. LifeGame is another example of cellular automata that involves densely populated 100x100 grid points. The live cells and dead cells are represented as values one and zero in the patch variable, respectively. For the optimized implementation of LifeGame and ForestFire in NetLogo (“NetLogo (opt.)”), the programs are written in a way that only relevant cells hold turtles; i.e., the numbers of turtles are much less than 10,000.

Let us take the LifeGame as an example. For each cycle in the Game of Life, there are three steps. In the first step called `calcNeighbors`, the turtles move around their home positions, accumulate the eight patch values in neighboring positions, and store the number into a turtles’ own variable. In the second step called `calcNextState`, the value in the variable is tested (in three conditional statements and two of them are nested) and one or zero is stored into the patch variable. In the last step, the system renders the patch variable onto the display screen. In these three steps, only `calcNextStep` gets the benefit from the predicated statement-wise model, because `calcNeighbors` and `calcNextStep` don’t contain any conditional statements. By comparing the numbers from LifeGame for SSW and PSW, one can tell that the micro-level improvements give a factor of 4 improvements (9.0 fps vs. 39.0 fps). This means that using the PSW model has practical value. Also, by comparing the number from PSW with the number from NetLogo, one can tell that the straightforward implementation on NetLogo (3.6 fps) is a magnitude slower than PSW (39.0 fps), and even with the optimized implementation (i.e., the implementation is forced to use indirect representation of the model), it cannot match the number from the PSW model. The ForestFire example exhibits the similar.

The GasTank example uses less number of turtles (2,000). With this many turtles, the ratio between the NetLogo implementation and Kedama is smaller (about a factor of 5). Even so, this shows that the implementations whose performance is not linear to the number of turtles reach the limit in real examples. (The result from the GasTank example for NetLogo (opt.) is blank, as there is no simple optimization that can be done.)

For the GasTank example, SSW outperforms PSW (93.0 vs 79.0). This is due to the fact that the data-dependency analysis is not precise; ideally, a read operation of a property of an object should not “depend” on a write operation to different property of the same object. However, the analysis only

treats each object as a whole, to mitigate the presence of special cases. As a consequence, there are cases when false data-dependency is detected, and the system sequentializes the statements involved. The number shown with asterisk (102.0^(*)) indicates the expected result from the precise analysis. (The number is measured by altering the analysis result by hand.)

As the absolute speed comparison, typical Kedama examples spend 60% to 80% of execution time in the C-compiled primitives written for Kedama. This means that it is not trivial to make the entire system, twice as fast as Kedama. Quite possibly, a dynamic instance-based object model implementation on such a language would add much overhead so that any resulting system would perform not much better than Kedama. Because Kedama allows dynamic modification of the code for the examples, we are allowed to say this is a reasonably good figure.

8.2.1 Further Performance Improvements

There are areas to gain more performance improvement. One of the biggest possibilities is to use hardware dependent optimizations.

One of the reasons to take the path toward the statement-wise models was the recent rise of General Programming on Graphics Processor Unit (GPGPU). The primitives for Kedama were written in Squeak and compiled with ANSI-C compilers. And, they perform the relatively simple task on homogeneous data arrays (32-bit `floats`, typically) in the inner loops. This is a close match with the GPU's functionality. Placing the data arrays in the GPU's memory and using the GPU's functions, we should be able to take advantage of modern GPU's computation performance.

More platform neutral optimizations can be done as well. One area is around the assignment statement into turtles' properties. Kedama computes the right-hand side values for all turtles, even though some of the values may be discarded because of the predicates. Ideally, the system could look at the predicates of the left-hand side first and skip the computation of values that would later be discarded. This would be a viable optimization but we have skipped this one due to simplicity.

Also, the number of the temporary arrays to store the intermediate values can be decided so it could be possible to avoid the allocation of such arrays. If the number of turtles in a breed doesn't change often, this would give about 5% performance gain.

For another possible improvement, the representation of the heading property could be changed. In the current implementation, the heading for a turtle is represented as a scalar in radians. In this implementation, the

`sin()` and `cos()` functions are called for every `forward` by action. This representation was used for historical reasons, but one could use a normalized vector (a pair of x component and y component) to represent the heading. (The Tweak version of Kedama uses this representation already.)

With the normalized vector, the `forward` by action can simply multiply the offset with the vector components and add them to the x and y coordinates. In this case, instead, the `turn by` and `setHeading` become more expensive; however, it is more common that a scalar angle is used for the argument of `turn by` value. In that case, the system can calculate the new components only once and use them for all turtles in the breed.

On the other hand, the heading value should be visible to the user as a non-intrinsic property. This requires inverse trigonometric function calls to convert the vector into an angle value. This, however, is needed only for the user-interface and less often than the actual use of the vector forms.

Chapter 9

International Deployment

The aim of the Kedama system is not only just being a research project, but also to gain wide acceptance at schools all over the world. To reach the goal, the author also implemented the multilingualization layer of Squeak [57]. Also, to adapt non-standard platforms that may be used in different regions in the world, Squeak VM has been ported to some exotic PDAs [58].

In this chapter, the work of the multilingualization and various VM ports done by the author are described.

9.1 Multilingualization

The original Squeak system only handled 8-bit Characters and Strings. This restriction was inherited from the original Apple Smalltalk-80 image. For the international deployment, this problem had to be solved. To solve the problem, the author started the implementation of a multilingualization (*m17n*) framework for Squeak in 1999.

Squeak is a “self-contained” system that models the entire computer. The *m17n* effort involves not just changing the representation of a string, but also all low-level functions such as keyboard input to high-level functionality such as text formatting and rendering, and compiling the code with such strings. A good set of design decisions had to be made to make it even work.

The goals of the design that the author set were as follows:

- The change to the system should be mostly transparent, i.e., as long as the clients of the `String` objects doesn't assume the internal representation, the client code should work without modifications.

- The system should not only be able to handle wide strings and characters, the author wished to be able to mix different languages at the same time in one text pane.
- While the internal representation can be decided by the author, there are already a wide variety of existing text encodings. The system should be able to communicate with other systems.
- Such `Strings` should be usable everywhere in the system. It should be possible to use them in not only the string literals, but also in class and method names if the user wants.
- The majority of strings in the system will be unchanged and can be represented in the same 8-bit characters and strings. For the sake of memory consumption and performance, the system should be able to use its original representation as much as possible.
- Different languages have their own text formatting rules. The framework should accommodate the different formatting rules that can be implemented by the native speakers of each language.

To satisfy these goals, the key designs that the Squeak m17n framework took were as follows:

- The system adds new class called `WideString` to represent a string that can store wide (32-bit) characters. A `WideString` has 32-bit quantities in it. However, the system uses `WideString` only when needed; namely, for a string that only has 8-bit characters in it, the original 8-bit representation is used. And, a `WideString` is implicitly used when only needed to store 32-bit characters.

The class for the original representation was renamed to `ByteString`. `ByteString` and `WideString` share the same abstract superclass called `String`. The old code that refers to the `String` class, and holds onto old `String` objects work mostly unchanged.

- The encoding of the characters and strings now use Unicode-based characters [59]. Unicode defines a 21-bit character set (20 bit + some fraction that can be covered by an additional 1 bit), and a code point is stored in the 32-bit slots in a `WideString`. The remaining 11-bit room in the 32-bit slot is used to store a Squeak-specific “language tag”. This language tag is used to discriminate the unified characters in Unicode, and provides language specific information such as rendering.

```
ArrayedCollection
  String
    Symbol
```

Figure 9.1: The original hierarchy around `String`.

```
ArrayedCollection
  String
    ByteString
    WideString
    Symbol
      ByteSymbol
      WideSymbol
```

Figure 9.2: The current hierarchy around `String` (indentation denotes the superclass-subclass relationship).

In an old implementation of m17n Squeak, the domestic code along the line of ISO-2022 and the “encoding tag” was stored in the 32-bit slots. Learning lessons from Mule and Emacs [60], the current Squeak m17n Framework switched to Unicode-based encoding.

- The rendering mechanism was re-written in Squeak. The interface is to render a given string onto given `From` object from a given point. It is possible to use an external library like Pango to render text. To support keyboard input, the system talks to the existing text input methods, rather than re-writing them. The `Sensor` is modified to handle such multi-byte character input from the VM, produces the right characters and sends them to the rest of system.

The class hierarchy around `String` in the original system was shown in Figure 9.1. The indentation depicts the superclass-subclass relationship. In the current system, the `String` was changed to be an abstract class, and new classes are added below it. The old instances of `String` were migrated to `ByteString`, and old instances of `Symbol` were migrated to `ByteSymbol`.

After the system was implemented, the eToys system was translated by Kazuhiro Abe and the Squeak community in Japan. With NHK TV

coverage and other publicity, eToys in the Japanese educational community took off, and started being used in many schools. Most notably, the Kyoto City Educational Board committed a long term project [13] in the Kyoto schools. In 2005, the work was incorporated into the official Squeak releases (6 years after the initial release of the m17n framework in 1999).

Other than the Kyoto schools, other educational programming language researchers adopted it and used it as the basis of their research projects.

9.2 Deployment over Various Devices

The Squeak VM is designed to be portable among various platforms. In 1998, the author ported the VM to perhaps one of the most exotic platforms that could possibly run Squeak.

The VM was later used in a research project called “The Parks PDA” at a Disney theme park [61] in 2000. The project investigated the concept of a theme park guide on a mobile device. The Squeak VM was used on a modified version of the Sharp Zaurus MI-C1 to run a customized application that provided graphics and sound rich media content.

Chapter 10

Conclusions and Future Directions

This chapter describes the conclusions of the dissertation and the direction to explore.

10.1 Conclusions

This dissertation has presented a programming environment named Kedama. In Kedama, the user can write programs to control thousands of objects in a graphical visual tile scripting language. The system allows the user to modify the program and data while the system is running so that the user can explore the problem domain by changing not only the parameters but also the program itself as well as the structure of objects. The interaction gives the user the ability to understand the described phenomenon deeply.

Kedama is written as an extension of Squeak eToys. By taking advantage of Squeak's dynamic nature and portable implementation, Kedama has been smoothly integrated into the existing eToys system.

One of the major goals is to give good performance, because handling tens of thousands of objects at a good enough frame rate is still some challenge on today's personal computers, and it is more so on the \$100 laptop and PDAs that are used in large scale around the world at educational scenes. We aimed to provide a system that can run at a speed that is comparable to a system written in C, while it allows dynamic modification and interaction.

To satisfy the performance goal, we have chosen a parallel execution model that we call the predicated statement-wise model. We put some

restrictions on what a turtle can do, and sacrifice language's orthogonality slightly. With this design and implementation, the system spends 60%-80% of its execution time of typical examples in the C-compiled primitives that have tight inner loops. This implies that the implementation is reasonably well optimized. The common examples written in Kedama often run an order of magnitude faster than the similar implementation in StarLogo or NetLogo.

There are many large and notable examples successfully written in Kedama. The examples include, parallel genetic algorithms, ant colony simulation, various pixel manipulations, gas diffusion simulations, as well as the examples explained in this dissertation such as epidemic transmission simulation, an ideal gas simulation, a forest fire simulation, etc. Such examples are written in a tile scripting system with (typically) dozens of small and meaningfully factored scripts. The experience suggests that the restriction on language is irrelevant in many cases, or can be worked-around with very little effort.

The user-defined scripts in Kedama are analyzed to extract parallelism. For the analysis, an attribute evaluator generator was implemented. The syntax-tree of the script is transformed in a manner of attribute-directed transformation and compiled into a code that takes advantage of the performance primitives in the execution engine.

The system is ready for actual users in classrooms. The phrases in natural languages that the user sees are fully-translated to various natural languages by using the multilingualization package for Squeak written by the author and his colleagues.

10.2 Future Directions

There are several immediate extensions to the current Kedama system.

10.2.1 Explore Different Languages

The current tile scripting system is our first attempt. The language is limited to simple and pre-defined arithmetic and the tile-scripting system practically limits the size of a program to 30-50 lines. This seems enough for writing typical simulations for educational purposes, but one would imagine the need for non-toy-like parallel languages. Toward making such extensions, more orthogonal semantics with strict SIMD semantics (transactional memory), explicit construct for specifying parallelism, and possibly the polymorphic SIMD model are interesting concepts to consider.

It should be noted that the an important aspect that we would like to keep is the dynamic interaction with the language, even with these features and extensions. Furthermore, an even more important aspect we would like to add is the language extensibility by the user. Toward this extensibility goal, my group is working to make a very, very micro-kernel language written in itself. By leveraging this work, we could make a user extensible programming language for parallel programming.

10.2.2 Transition Between “Big” and “Small” Objects

There is a big problem remaining that was reported by Resnick. And, Kedama still doesn’t provide a good solution for it. Many students seem to be able to handle one to a few objects, but cannot pass the bridge to the massively parallel objects. The reasoning was that there is a psychological gap between handling a few “big” objects and many, many “small” objects.

Solving this problem will be a big next achievement. A possible approach can be summarized as follows:

- Get rid of Kedama World, and allow turtles to reside in the World.
- A big object (which the user usually begins with) can have different appearances in several levels of details. The lowest detail would be a dot, but having a set of pre-calculated rotated bitmaps, and an arrowhead like bitmap can be there in-between.
- A big object can also have the equivalent of siblings in eToys. Siblings share the same shape and behavior.
- With a trigger by some user action or system profiling, the set of siblings convert their internal representation to possibly the “horizontal” representation described in Chapter 7. These big objects can still appear the same on screen, but the accessors of the basic properties is modified so that the value is fetched from and stored into the arrays. The existing scripts are “vectorized” and put to the equivalent of the “breed” object.
- By the user’s blessing, the big objects (with converted internal representation) can change their appearance to lower-detail one. In the other words, the user can make the transition between them.

By making the transition as smooth as possible, we hope that the user could see the small ones as the “same” objects as big ones, and could still re-

late himself to *one of* the objects. By keeping the horizontal representation, the performance can be still comparable to current Kedama.

To phrase it differently, the user can start with an ordinary object and *later* convert it and its siblings. This should foster more exploratory style programming.

10.2.3 More Types for Patch Variables

A patch variable in the current implementation only stores non-negative integers. There are already a few commands that can interpret the content as ARGB color values, it is still not a sanctioned data type. At least, the patch variable should be able to store simple types such as colors, signed integers, and floating point numbers, but also 2D vectors.

With 2D vectors in a patch variable, new classes of examples can be described. An examples with vector field is an obvious one, but also more elaborate collision detection, where the particles' speed is represented as vector, and the sum of the collided particles can be used to preserve the energy.

With the patch variables in different types, the type checking mechanism of the tile-scripting system should be modified. For example, the user should be able to change the type of a patch variable that is already used in an existing script. The system should accommodate the type-error gracefully, as it does for properties of scalar objects now, and give the user to proceed with it.

10.2.4 Multiple Worlds Work Together

In the current system, a breed of turtle is bound to a particular Kedama World when it is created. If we get rid of this restriction in the following way, there, again, can be new classes of examples can be written.

- Make it so that not only the eToys World, but also any container such as a “playfield” or “holder” can support turtles.
- Add another property to each turtle that represents the container of it. This property should be accessible in the tile-scripting system. Also, each container knows the breeds that may be in it.
- Upon rendering a container, it consults the breeds and ask them render the turtles. Each breed, in turn, checks the container property of all turtles in the breed and ask them to render. This “selective” action can be unified with the predicated action.

- All turtles in the breed execute a command for the breed. However, the action to take at the edge can be different. Specifically, a turtle should be able to “migrate” to another container by assigning to its container property.
- Through some user interface, the user can layout multiple containers and their “connectivity”. With this, a container should be able to answer the neighboring other containers.

With this feature, the user can write, for example, a electric circuit with containers representing wire and turtles representing electrons.

10.2.5 3D

To write some interesting physics simulations, having the third dimension is often helpful. In a 3D world, each turtle would have to have the position vector and the rotation matrix to represent the direction. Alternatively, a compromised representation with two 3D vectors (ignoring the rotation) may be suffice for wide variation of examples.

One thing to note that the concept of the patch, or discrete cells in the space, would consume prohibitively large amount of memory. (100x100x100 cells may consume a few megabytes.) The “fields” idea in the following would be a solution for this problem.

Whether the new system will be implemented in Squeak or not, Squeak and its variation are the basis of 3D collaborative environment called “Croquet” [62]. By implementing the 3D version of Kedama on top of Croquet, the user will have a way to share the projects and collaborate with other users over the net.

10.2.6 Fields

In the current implementation, a turtle’s coordinates are rounded to access a patch cell. This prevents from writing some examples that require precision, and also requires more memory.

We would like to add the concept of field that is somewhat similar to the Computational Field Model by Tokoro [63]. A field object holds an expression or a function of coordinates in floating point numbers and returns a value (should be in any type). To the user, it can look similar to the current patch variable; the turtle itself will be the implicit argument of the function and its coordinates is used.

On the current eToys system, an arbitrary expression will require the interpretation in the Squeak level, and may cause the performance problem. For this problem, an object engine with dynamic compilation facility will be helpful.

10.2.7 Hardware Dependent Optimization

With the help from hardware, more optimization is possible. One of the reasons to take the path toward the statement-wise models was recent rise of General Programming on Graphics Processor Unit (GPGPU). The primitives for Kedama perform relatively simple task for homogeneous data arrays (32-bit floats, typically) in the inner loops. This is a close match with the GPU's functionality. Placing the data arrays in the GPU's memory and use the GPU's functions via shader languages, we should be able to take the advantage of modern GPU's computation performance. This would save the transferring large chunk of data through the main memory bus. At the same time, eToys and Kedama rely on reading and processing data in many occasions, so there will need some mechanism for it.

Utilizing SSE and MMX instructions will give us substantial gain in performance. The inner loops of primitives can be re-written with such instructions. While the data has to go through the memory, the primitives would run 2-3 times faster.

Furthermore, the memory location of homogeneous arrays can be aligned by modifying the memory management system. In the current system, all the data is only 4 byte aligned, but 16-words alignment will help for writing logic with such instructions.

Appendix A

Squeak

Squeak[1] is a dynamic object-oriented programming system. The programming language is designed to be general-purpose and its implementation has a lot of interesting characteristics that help the implementation of Kedama. Also, there is an educational environment called Squeak eToys (simply referred to as “eToys”) implemented in Squeak that was used to be the basis of Kedama interface.

In this chapter, the programming language and environment aspect of Squeak is explained.

A.1 Squeak: Its Origin and Concept

Squeak was originally created to be an implementation system for end-user educational software. The creators of Squeak wanted it to run on wide variety of platforms, and to be flexible enough allow dynamic modification to itself.

What was slightly different from other many environments is that the Squeak creators have longer experience than almost anybody else in the research of programming environment as well as the educational software. Some of the authors, Alan Kay, Dan Ingalls, and Ted Kaehler had implemented the original Smalltalk [64] and whose primary purpose was an educational environment for children, drawing upon the great lessons of Logo by Papert. Of course, Squeak inherits the ideas of Smalltalk but provides a modernized implementation.

One way to look at Squeak is that it is a software environment that models the entire computer by a single, uniform concept. In this concept, a computer system is considered to be constructed from smaller, but equally

capable, sub-computers. Such sub-computers (and sub-sub-computers and so on) are called “objects”. The objects, or computers, are connected with each other via network and able to send “messages” to each other to carry out computation. When a message is received by an object, the object invokes a “method” that describes what to do. A message can carry arbitrary number of arguments, and the “receiver” can use the arguments in the invoked method. To refer to the idea of “objects and messages”, a term “object-oriented” was coined by Alan Kay. One of the key ideas was that because the sub-computers are also computers, the parts have the same independence and power as the whole.

However, during the adaptation of the term, the key concept of original idea was somewhat “dispowered”, and the boundary between it and similar but different idea of “abstract data types” was blurred in the industry and research community. The abstract data type, that has closer ancestry in data and procedure paradigm, were often considered that “the part is weaker than the whole”. While this may be more accepted “definition” of object-orientation, the author tries to follow the original definition of the inventor.

Also, the design principle of the Smalltalk system, which is the direct ancestor of Squeak, is nicely summarized in [65]. This article discusses the programming language and computer system from the point of view of human and computer interaction.

A.2 Squeak Language

In the Squeak language, the programmer specifies what happens when a message is received by an object. The way specifying the behavior of an object is to write “methods” in terms of a sequence of message sending when a certain type of message is received. The textual language of Squeak is originated from Smalltalk-80. (Hereafter the term Smalltalk is used to refer to Smalltalk-80, otherwise explicitly stated.)

A.2.1 Squeak Syntax

Since the only concept to describe the computation is “a message sent to an object”, there is only one abstract syntax form needed; the form to specify a message sending. Namely,

```
<receiver> <message selector> <zero-or-more args>
```

However, to improve its readability, Smalltalk and Squeak has three variants in its concrete syntax. The first form, called “unary message”, is used for

a message with no argument; it just specifies the receiver and the selector. For example,

```
anObject name
```

sends `name` message to `anObject`. Usually, `name` returns the human readable name of the object as a `String` object. The second form, called “binary message”, is used when the message selector is intended to be an arithmetic operator. A binary message can have only one argument, and the selector is (usually) restricted to the non-alphanumeric characters. For example,

```
3 + 4
```

sends “+” message with argument “4” to an object named 3. Usually, a “+” message returns the sum of the receiver and the argument. The third form, called “keyword message”, is used to specify more than one argument to a message. The arguments are tagged with the keywords that annotate the meaning of the arguments. For example,

```
'abcde' copyFrom: 2 to: 4
```

sends “`copyFrom:to:`” message with 2 and 4 as arguments. A string surrounded by single-quotes is a string literal (an instance of `String`), and it is indexed from one in Smalltalk. Therefore the above message returns `'bcd'`.

A.2.2 Messaging

The term “return” in previous section may need some explanation, as the concept of messaging doesn’t necessarily mean there is a reply. In fact, one of the ideas when Smalltalk was still in its incubating stage was “one-way messaging”, in which the “result” is sent from the receiver to the sender in separated and explicit reply message. (The idea influenced the Actor model by Hewitt [66][67].)

In Smalltalk and Squeak, a more conservative approach was taken. All the message sending returns a value (an object), as if it were a form of function call. In other words, the sender waits and synchronizes until the receiver returns the result, or when a “message expression” is evaluated, the return value becomes the result of the expression.

A.2.3 Classes in Squeak

Smalltalk and Squeak is a class-based object-oriented language; i.e. each object belongs to a “class”, that specifies the internal structure and the

behavior of the object. Typically, more than one object belong to the same class; i.e., a class can be considered to provide a grouping of similar objects. The objects that belong to a class are called “instances” of the class.

A class can “instantiate” instances. The internal structure of an object, often referred to as the “shape” of an instance, is specified by a dictionary of names and corresponding slots, in which references to other objects can be stored. Such slots are called instance variables. The instances of the same class have the same set of instance variable names, but each instance can have different set of values. Sometimes, it is convenient to have methods that read or write an instance variable; such methods are called “accessors”, or “getter” and “setter” of the variable.

The behavior of an object is specified as a dictionary whose keys are “method selectors” and the elements are the body of executable methods. When a message is sent to an object, the name of the message, or “method selector”, is used to index the dictionary of methods and proper method body is invoked.

Classes are organized in a tree structure that is called “class hierarchy”. In the class hierarchy, a class has a “superclass” as its parent, and may have some number of “subclasses” as its children. In the Smalltalk-80 tradition, a class called `Object` was at the root of class hierarchy; i.e. all objects are instances of transitive subclasses of `Object`¹

A child class (transitively) inherits the instance variables and methods from its superclasses. The set of instance variables of a class consists of all instance variables defined in the chain of superclasses and the ones defined at it. Note that a subclass cannot define an instance variable that is already defined in its superclass chain. On the other hand, a class can provide the method in the same name that its superclasses define. This is called “method override”. An overriding method provides modified behavior. Since it provides the same set of methods as its superclass, an instance of the subclass can be used anywhere as the one of its superclass, without modifying the other part of code. The fact that class hierarchy is a tree means that a class can only have one superclass. This makes Smalltalk a “single inheritance” object-oriented language.

¹In later Squeak, there is “a roof over the roof”, and a class called `ProtoObject` that provides the unique root for proxy type (non-standard) classes. But the detail is out of scope of this paper.

A.2.4 Everything is an Object

Smalltalk is a “pure” object-oriented language; it means that everything in the language is an object (a sub-instance of class `Object`). The “meta” concepts, such as a class and variable bindings are represented as objects. Furthermore, the data structure that are used for the computation, such as methods, “contexts” (the execution stack frames) and the compiler and related data structure such as parse trees, are also objects. This uniformity helps in programming, especially when manipulating the language itself.

The concept of objects and messages matches well with these meta-concepts. A class, for example, should be able to respond to requests such as making new subclass, creating an instance, add/remove a method and an instance variable. Such requests can be represented as messages straightforwardly.

Another important aspect of Smalltalk is that the rich “protocol” an object understands. In Squeak system, `Object`, as the superclass of all objects, provides as much as 400 methods. This is a surprisingly larger number compared to other typical object-oriented languages; for example, Java’s `Object` has 12 or so public methods. The fact that an object in Smalltalk can understand the rich protocol including reflections, printing and reading, triggering events, etc. makes objects very powerful entities.

On the other hand, some consider it harmful to have hundreds of methods that serve different “aspects”. The original vision of an object-oriented language was not to restrict to single inheritance, but to incorporate an idea called “sideway composition” [68] such as `PIE` [69] and later `Traits` [70]. The sideway composition idea hasn’t materialized fully, but Ruby’s `mix-in`, and now `Traits` are adapted by Perl and other languages provide the implementations. The Smalltalk language is still evolving in interesting direction.

A.2.5 Dynamic Modifications

Another important aspect of Smalltalk is the wide area where various dynamic changes are allowed. Because the power of dynamic modification is more apparent when it is in the whole Squeak environment, it is often neglected in “reductionism” analysis of programming languages.

An example is to add an instance variable to an existing class. If the user requests that, the slot for the instance variable is added to all existing sub-instances of the class. Most of object-oriented systems used today don’t allow such class shape change at all, or the existing instances don’t follow the

change. Such approaches make sense only when there is something “outside” of the system. In a self-contained system, *any* part of system should be changeable, and it is more preferable if it can be done in a uniform way.

Such a modification is also modeled as a message sending. For example, adding an instance variable to a class is done via a message sent to the class. Also, defining a new class under a superclass is done via a message to the superclass. Writing a piece of program involves defining a class and methods, so such dynamic change to the system *is* writing programs. At the highest level, the message to the computer is sent from the human. This full closure of the model described in the paper “The Design Principle of Smalltalk” [65] is the basis of the philosophy of the system.

A.2.6 Context Aware Exceptions

In regards to the nature of the execution, the semantics of exception is also worth to mention. A typical model of exception in other languages is that it travels toward bottom of stack to find a matching exception handler. Stack frames between the exception handler and the stack top is discarded and the control of execution jumps into the exception handler. On the contrary, an exception in Squeak “remains” at the top of stack by default, and the different kinds of exceptions can provide different implementations to specify the way the exceptions are handled. Such an implementation can utilize the fact that contexts (stack frames) are first class object, and the exception handlers can be written in Squeak itself.

For a normal type of exception, the exception traverse through the calling contexts and looks for the appropriate handler for itself. If it finds one, it jumps to the handler and continue the execution from there. Only at that time, the stack frames above the handler get discarded.

On the other hand, an implementation for some kind of exceptions can modify the related data and contexts and “resume” the execution where the error occurs, as if there were no exception occurred. It is useful to supply the necessary data in lazy manner, for example. Or, another handler could make up a return value by itself, and return it to the caller of the context as if it was returned from the called (and error-causing) context.

If no handler is found in the stack frame, instead of the entire system dies (perhaps with minimum stack dump), the Debugger window is opened. The use of Debugger is described further in section A.3.

This manner of exception handling wasn’t unique to Smalltalk itself, but the languages in Lisp family do the similar thing. However, sometimes a good idea doesn’t get carried over to the later languages. The Java language

and Eclipse development environment have been trying to be more flexible, and from the user's view many areas are getting covered.

A.2.7 Collections

Squeak has a rich collection library. A class in a collection library represents a collection of objects with different characteristics. Some of such characteristics includes: indexable or not, the type of index (indexed by number or any objects), the equality criteria of objects in collection, resizable or not, homogeneous or not, etc. As long as an object provides the protocol that satisfies the ordinary collection protocol (defined at the abstract class `Collection`), the internal structure of the object doesn't matter. For example, the class `OrderedCollection`, which represents, a commonly used collection for a number-indexable, resizable, heterogeneous collection, internally holds an instance of `Array`, which is fixed size variant, and overrides the basic indexing methods accordingly.

The above means that there is circularity, as one collection class can be represented with other collection class. Where is the fixed point of the circularity? The answer is that there are classes more essential than others, and the most essential classes for collections are "variable length" ones. An instance of variable length class can have slots that are indexed by integral numbers, not by names. The number of slots can be specified upon the instance creation and it cannot be changed once it is created. In other words, from the implementation's point of view, the layout of an instance of variable length class looks like uniform size slots laid out in continuous memory region. Other instances that require some storage area can use the variable length instances to represent themselves.

An `Array` object stores the references to objects in the slots, and it is often used as the basic "data structure" for other objects. Another notable example of variable length class is the `ByteArray`; whose instances store the 8-bit values in the slots.

A.3 Squeak Environment

A.3.1 Computer as an Object Environment

Smalltalk is also known as a programming environment. This means that not only the language elements described in A.2, but also the computer peripherals are provided as objects in the system. For example, there is an object called `Sensor`, which represents the mouse and keyboard. When

a physical key on the keyboard is hit, a corresponding event (that is an object, too) is generated and sent to other objects from `Sensor` as a message. Similarly, the physical computer display screen has a model in the system; an object named `Display` behaves as if it is a normal 2D graphical object, but when a portion of it is modified, the change is reflected to the physical screen. It is in fact a subclass of normal graphical object, so that not only one can modify the pixels in it, they can be read out in the same manner the user and programmer would do with a standard graphical object.

Again, this means that the programmer and system can ensure the appearance on the screen is *exactly* what it is programmed to be.

A.3.2 Development Tools

The advantage of having everything in the language and environment cannot be exploited if there were no tool for the user to access them easily. The basic idea of providing a set of tools in Smalltalk and Squeak is to let the user “talk” to any object in the system in a uniform manner.

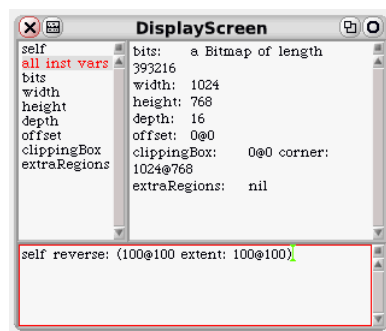


Figure A.1: An inspector on the `Display` object.

The “conversation” with an object is enabled via the standard text editing tools that Squeak provides. A text tool lets the user edit text in it, but also the selected part of text can be evaluated as an expression. When the expression is evaluated, the pseudo-variables are automatically bound to proper objects; namely, “`self`” is bound to the receiver, and `thisContext` is bound to the current top stack frame. With this trick, the references to receiver’s instances variables or temporary variables of the stack frame are accessible from the evaluating code. In figure A.1, the “Inspector” tool for the `Display` object is shown, in the bottom pane, the user can type expressions and evaluate it to test the object’s behavior. In an Inspector `self` is

bound to the inspected object. An inspector also shows the list of instance variables and shows the value of selected variable.

The System Browser, in which the user writes the code, is another example. The System Browser organizes the classes in categories, and let the programmer navigate through the class and method cross-references, as if they were hyper-linking, and edit the code. When the code is edited and “accepted”, the Browser and Compiler do reasonably good compile-time error checks on the method, and report them interactively, so that the user can fix the problems immediately.

More notably, the Debugger tool is the essential part of the Squeak development environment. As written in section A.2, when an exception occurs, the exception can describe how it wants to be handled. The default action for an exception is to start the debugger with the chain of context that caused the exception.

In the Debugger, the user can not only investigate the content of stack frames or related objects, but also he can rewrite the method definition in the debugger, as the debugger is also a variant of the same text editing tool. What will happen if you edit the code which is being executed (and stopped at a breakpoint) and “accept” the code? Following the design philosophy [65], the new code takes the effect *immediately*; while keeping the context chain below the method just have been edited, the edited method gets “restarted” from there, without needing to scrap the result from the execution led to the point.

The Debugger also lets you to easily debug many “branches” of execution. If the programmer encounters a bug in the middle of code, the programmer can spawn another execution thread from the point of execution, for example.

A.3.3 GUI Frameworks

Since the programmer has the control over the abstract input devices and the screen, it is easy to write a GUI framework. Squeak’s official distribution has two GUI frameworks that are called Morphic and MVC. MVC is a traditional GUI framework (in fact, arguably it is the first GUI framework). Today’s users and programmers, however, use a modern GUI framework called Morphic in Squeak. Morphic [71] was originally developed for Self programming system [72], and later ported to Squeak by one of the original author (John Maloney). Morphic allows direct manipulation, non-rectangular objects and smooth resize and rotation of graphical objects in a “two and half dimension” field. It also has an elaborate time-based event triggering mechanism

so that the burden of multi-thread programming, which is often required in graphics program, is largely alleviated from programmers.

Other than these two official GUI frameworks, many special purpose GUI frameworks have been written in Squeak. One notable example is PenSprites framework done by the author's group [61]. PenSprites was a GUI framework designed for a pen-based operation on resource-limited hardware such as a PDA. Handful base classes and a few hundreds of methods was enough to make a light-weight GUI framework that allows non-rectangular widgets, time-based animation and hyper-link interaction.

All of the visible elements in the Morphic GUI framework are sub-instances of a class called *Morph*, and organized in a logical tree structure. A *Morph* object is capable to draw itself, to react to the events including user interaction and time-based event triggering, and to manage the parent and children relationship. At the root of parent-children relationship, there is a (sub-)instance of *Morph* called *World*. Again, having the transitive parent of all visual components on screen in the environment helps to control the appearance and behavior of program precisely.

While a *Morph* is conceptually a simple object, similar to the richness of the protocol that an *Object* understands, the class *Morph* holds *many* methods. In Squeak version 3.8, *Morph* has over 1,200 methods, including the methods to control the many rendering embellishment options, pen-based interaction hooks, and various interface to tile scripting system described later.

Even though, one could say that the core of Morphic is simple. Once it is up and running, the Morphic kernel repeats a simple cycle of interpreting input, time-based event processing, and screen update. The input event processing checks the events from the *Sensor* and dispatches them to the proper *Morphs* in the *World*. The recipient *Morphs* invokes specified methods. Also, in each cycle, the timer is checked to see if some registered events should be fired as the timer elapsed the scheduled time. The notification of timer is delivered to the *Morphs*. If these events causes the *Morphs* to want themselves to be redrawn, the *Morphs* request what area in the screen should be redrawn (the area is called a "damaged area"). At the end of cycle, the *morphs* in the damaged area are asked to draw themselves on screen.

In summary, the advantage of Morphic GUI framework is its uniformity and simplicity, time-awareness event mechanism, and reified system objects.

A.4 Execution Model and Implementation

In this section, the implementation of Squeak is explained.

A.4.1 Execution Model

The Squeak environment runs on the Squeak virtual machine (VM). The virtual machine provides a well-defined “surface” that looks like abstract hardware on which the Squeak code are executed on. On the VM, a memory image of running Squeak machine, or Squeak Virtual Image (*VI*), is loaded and executed. The image contains objects of Squeak system and new objects are created to carry out the computation, and objects no longer needed are reclaimed. Since the virtual machine provides a well-defined virtual platform, the image can be used as is on any platforms on which the VM can run.

In the following, the detail on VM and the VI are explained.

A.4.2 Squeak Virtual Machine

The Squeak environment runs on the Squeak virtual machine (VM). The Squeak virtual machine is, typically, an application on an operating system, but it can also run directly on the top of hardware. The Squeak virtual machine offers following features for Squeak execution:

- A well-defined instruction set that is suitable for Squeak code execution.
- A stack pointer, instruction pointer, and current context pointer, and etc. for executing instructions.
- An object memory management system.
- Access to the common hardware peripherals, such as keyboard, mouse, and display.
- A set of C functions that provides better performance for computation intensive operations.

In other words, the virtual machine almost looks like (therefore it is virtual) a hardware that is designed to execute Squeak code.

A.4.3 Squeak VM Instruction Set

The instruction set of Squeak VM is very close to the one in the “Blue Book” [64]. Most of the instructions are encoded in one byte; that is why they are called “bytecodes”. Reflecting the stack machine architecture and semantics of Squeak, the instruction set mostly consists of manipulation (push and pop operations) of stack, sending messages, and accessing the slots of objects. Because a context is an object, accessing the local variable is very similar to accessing the slots of a normal object. Most of the instructions are one byte long and a few take the following byte as their extended argument.

A sequence of bytecode for a method is stored in the instances of a class called `CompiledMethod`, which is a variant of `ByteArray`. Since it is a variant of `ByteArray`, the instructions are ensured to be stored in consecutive bytes in the physical memory. It opens up the opportunity to execute the code reasonably efficiently. Also, a context, or stack frame, is an `Array`-like object (a sub-instance of `ContextPart`), as stack frames should be able to store arbitrary objects in its slots.

The instruction pointer (IP) and stack pointer (SP) in the VM are used to address the next instruction to be executed and indicates the current stack top. The IP and SP point to a memory location within a `CompiledMethod` and `ContextPart` objects, respectively. The values are visible and modifiable in a restricted way from the program in the Squeak Virtual Image.

A.4.4 Squeak Memory Management

The memory management system governs the object allocation and deallocation (automatically collects unused objects as garbage). Since the VM is designed to be an abstract platform for executing Squeak program, what the memory management system allocates are the objects in Squeak preferred format, and does the garbage collection based on the object format.

Squeak’s garbage collector is a two-generation, mark and compaction collector. The young space collector, which is also called “incremental GC”, is invoked every 4,000 object allocations by default. As shown in [73], most of newly allocated objects die early so that the mark phase and compaction phase can be done in short time. When the Squeak’s garbage collector was designed in ’96, typical incremental GC took 5ms or so on a typical personal computer. On a typical computer today (2006), it usually takes less than 0.5ms. This is short enough not to disturb soft real-time process, such as movie play back or sound synthesis. Note that Squeak’s collector was designed for small devices in mind, so it doesn’t use any semi-space

even for young space. There is a data structure that holds the objects in the old space that has a pointer to an object in the young space. Upon compacting the young space, the objects in the young space and the root table are scanned and the pointers to moving objects are adjusted. The pointer adjustment is done by using the “pointer-reversal” technique [74], in favor of the space efficiency and not having any fixed-size table in VM.

4,000 is a small number for allocation count to trigger the incremental GC, especially on today’s processor performance. For server type applications, it is often increased by the programmer’s choice. However, the concept of doing fast GC frequently (dozens times per second) has a proven record of success.

As for the memory format of an object, there are several different formats for different objects. The `SmallIntegers` are represented as one word (32 bit) and tagged with 1 bit at the least significant bit, therefore they can represent from -1073741824 to 1073741823 (-0x40000000 to 0x3FFFFFFF).

A usual object (i.e. non-`SmallInteger` object) has the object header. Object headers are variable length of 1 to 3 words. The header describes the object’s size, the flag for indexable object, class, and intrinsic hash value. But, among others, the *type* of slot is the most important information.

The slot type for an object is either pointers, non-pointer words, or non-pointer bytes. For a pointer slot, the direct pointer to another object’s memory location is stored. For non-pointer words, the 32-bit word in a slot can be used without the integer tag bit.

It is useful, in terms of the interoperability with C language, to store the data in the same bit representation as C’s (`unsigned`) `int` or `float` (on ILP32 architecture model). If the representation is compatible, a primitive (a routine compiled by a C compiler) can seamlessly access the slots, and perform arithmetic operations with no overhead. It is also true for a non-pointer bytes object. For this purpose, there are a wide variety of Squeak classes that have non-pointer word formats and store the data compatible with C language. The ones that are used in Kedama are `IntegerArray` class to represent the array of `signed int`, `WordArray` for `unsigned int`, and `FloatArray` for `float`.

Non-pointer objects are relief for the garbage collector, because the garbage collector skips the mark phase for such objects.

A.4.5 Accessing the Platform Functions

The VM provides the standardized access to hardware peripherals from the Squeak Virtual Image. In a sense, these peripherals are “wrapped” and

can be seen as Squeak objects. This allows the code in Squeak to access such objects in a uniform way. The same paradigm of “objects and message sending” can be used even with such wrapped objects. For example, the hardware display screen itself is treated as a graphical object so that client code can write and read the content of the screen as if it were the same standard type of graphical object.

To absorb the difference of actual platform the VM is being ported, the VM porter implements the “support code” in C programming language. For the minimum implementation, the porter only needs to write the C code that specifies how to 1) send a bitmap data in Squeak to the physical display, 2) map the events generated for keyboard and mouse to events in Squeak and 3) access the file system. Optionally, the support code for input and output of sound from/to the sampled sound, TCP and UDP network socket I/O, serial port I/O, etc. are provided to support the Squeak’s standard I/O.

A.4.6 VM generation and Performance Primitives

Like any other systems on virtual machine implementations, the biggest advantage of Squeak VM is its portability. However, Squeak pushes the idea into extreme. The only requirement for the virtual machine to run is that the platform is capable to allocate a reasonable size (from a few mega bytes to 2 giga bytes) of a 32-bit addressable flat memory region and do the 32-bit integer arithmetic. (Having floating point arithmetic is preferable, but they can be emulated by software.) Squeak has been ported to more than two dozen platforms, including the “major” ones such as Microsoft Windows 95 to XP, Mac OS 7 to X, almost all Unix variants on different architecture, and many different PDAs [75].

The way Squeak achieves the portability is to write as much part of the implementation of VM as possible in a high-level language; in fact, in Squeak itself. The VM code can be run, or simulated, in the Squeak virtual image so that it can be tested and debugged dynamically with all Squeak development tools. Then, the code is translated to C from Squeak code to achieve reasonably good performance. The code is written in a way such that it only contains the control structures and types of Squeak objects that are easily translatable to C. The core interpreter and the garbage collector are written in this way.

Other than the core interpreter and garbage collector, the programmer can add custom routines that are called “primitives”. Such primitives are either platform-dependent, which calls the platform API so that it cannot be easily migrate to another platform, or platform-independent, which doesn’t

call such API. Typically, the platform independent primitives are also written in Squeak and translated to ANSI C, and linked together with the VM.

An interesting view of the dynamic behavior of the Squeak execution is to think about the dynamic call tree of all method invocation. Because the model implies that every method is described in terms of other message sending, there is no fixed point in this recursive method definition. To cause real computation, there must be something that doesn't send any message but carries some computation out by itself. In Squeak, they are the primitives.

A.5 Squeak Virtual Image and Object Format

The Squeak Virtual Image is a platform independent memory format to represent the whole group of objects. As written in previous section (A.4.2), the only requirement for the format is that the objects can be laid out in a 32-bit addressable continuous memory region; the same image runs both on little and big endian platform. (While the 64-bit version of image format is available, it is not used for Kedama project.)

Once the VM started running, the memory region of size that is enough for a VI is allocated and the VI is loaded into the memory region.

Since the garbage collector does the compaction, most of the live objects tend to occupy adjacent area in memory. Newly created object is allocated after the object at the highest address.

The content of virtual image can be written out as a file. This action is called "taking a snapshot", as it captures all the objects as is and make them resumable for later sessions. The header of the stored file contains the base address of the previous session, so when the snapshot is loaded into new session, the offset of the old base address and new based address is calculated and all the direct pointers in the image are adjusted.

Again, the virtual image format is platform independent. For any platform that the VM can run correctly, the snapshot can be resumed, no matter where the snapshot was created (i.e., on little endian or big endian computer).

In Squeak, there are some other means to save the group of objects into a file. The most important ways are called "ImageSegment" and "SmartRefStream". An ImageSegment is a subgraph of the entire object graph in the image. The subgraph is specified by defining the "boundary" of it, and created by traversing the all reachable object within the boundary.

This traversal is very similar what the garbage collector does, so it is ac-

tually implemented in a garbage collector. It artificially marks the objects at the boundary, then start the marking phase from the root for the subgraph, the objects in the subgraph are marked, but not the ones in outside. The marked objects are then serialized in a way to keep the internal references and stored in a file (this is called an `ImageSegment`).

While `ImageSegment` is very fast, it turned out to be fragile. Class shape changes in the image breaks the compatibility of exported `ImageSegments`. `SmartRefStream` was written to overcome the problem. `SmartRefStream` traverses the objects in the similar way that the garbage collector does but it is written completely in Squeak. A class shape change of exported objects is adjusted upon loading.

Appendix B

The List of Features of Kedama Objects

The work of this dissertation, Kedama, is built as an extension of eToys. In this chapter, the extension is described. In addition to the objects of eToys, Kedama defines three new kinds of object. The one that plays the central role is the *parallel turtle*. A parallel turtle represents a group of homogeneous turtles, and often explained as “a breed of turtle”. The second type of object is called the *Kedama World*, representing the place in which the turtles reside. A Kedama world is a two-dimensional plane and provides the coordinates and headings for the turtles in it. By default, the logical extent of a Kedama world is 100x100. The last kind of object is the *patch variable*. A patch variable provides a two-dimensional matrix of cells. Each cell of this matrix corresponds to a grid point in the coordinates of a Kedama world and holds an integral value.

In the following, the commands and properties of these new objects are explained.

B.1 The Kedama World Object

Table B.1 lists the Kedama World specific properties. It shows the names of the properties and commands, and their types. In the “Type” column, “(r)” denotes the property is read-only. Some of the properties have both getter and setter and allow reading and writing, while the others only have getter and allow reading.

The `color` property specifies the default background color. As shown in Figure 3.1, the patch variables and turtles are placed “on” the Kedama

Table B.1: The properties and commands of Kedama.

Name	Type
Turtle properties (intrinsic)	
x, y, heading	Number
visible	Boolean
color	Color
Turtle properties (non-intrinsic)	
patchValue	Number (r)
upHill, angleTo, distanceTo	Number
getReplicated, turtleOf	Turtle (r)
Turtle commands	
forward by, turn by, die	
Turtle breed properties	
turtleCount	Number
Kedama World propertie	
color	Color
pixelPerPatch	Number
topEdgeMode, bottomEdgeMode, leftEdgeMode, rightEdgeMode	Symbol
patchDisplayList, turtleDisplayList	Collection
patch variable properties	
color	Color
displayType	Symbol
shiftAmount, scaleMax, sniffRange, evapolationRate, diffusionRate	Number
patch variable commands	
decayPatchVariable, diffusePatchVariable, clear	

World. When rendering a Kedama World, it is first filled with color property and then the other objects are rendered on it.

The `pixelPerPatch` property provides the scale factor when the Kedama World is displayed on screen. For example, if the value is 2, a Kedama world that is 100x100 in logical size is rendered as a 200x200 square on screen. The default primitive in Squeak that handles the bitmap scaling is generic and multi-purpose one, but not specialized to scale with integral magnification. The value of `pixelPerPatch` is used by a customized primitive that only handles the integral magnification value.

The four “edge modes” specify the action taken when a moving turtle hits the top, bottom, left or right edge of the Kedama World. The possible choices for them are: `wrap`, `bounce`, or `stick`. The meaning of these is discussed later with the `forward by` command of turtles.

The `patchDisplayList` of a Kedama World holds the list of patch variables associated with it. When the Kedama World is being rendered, the patch variables in the list are alpha-blended in the specified order. The way the values in a patch variable are converted is explained in the patch variable section.

Similarly, the `turtleDisplayList` of a Kedama world holds the list of breeds of turtle associated with it. When the Kedama World is being rendered, the turtles are overlaid in the order. The way the turtles are rendered is explained in the turtles section.

B.2 Patch Variables

There are three commands for a patch variable that do the bulk mutation of its cells. The `decayPatchVariable` command reduces all values at the rate specified by `evapolationRate` property. Similarly, the `diffusePatchVariable` command reduces all values, but it uses the average of the neighboring cells for the new value of a cell. By averaging neighboring cells, a cell that has large value gets smaller value and effectively spread the value from the cell. The `clear` command clears the values in all cells to zero.

When a patch variable is rendered a Kedama World, what happens is that the value in each patch cell is converted to a color by a function and the color is alpha-blended into the corresponding pixel in the Kedama World.

Three different functions are provided for this integer-to-color conversion. A property called `displayType` specifies the function. The possible values for the property are `logScale`, `linear`, and `color`. If `displayType` is `logScale`, the logarithm of the value is calculated first, and the log value is used as the

saturation, or alpha channel value, of the hue specified by the `color` property of patch variable. The `scaleMax` property specifies the cut-off value. Let's take an example where a patch cell has 500, the `scaleMax` is 1000, and the `color` property is `Color red` or `0xFFFF0000` in the `A:R:G:B=8:8:8:8` format (i.e., each of the alpha, red, green blue components of a color is specified as 8-bit quantity). In this case, $\ln(1000)$ is 6.91 and $\ln(500)$ is 6.21, so the saturation value for the cell is $6.21/6.91 = 0.90$. This value is used as the alpha blending factor and color value is blended into the Kedama World's graphical representation. In this example, 10% of the Kedama World color and 90% of red are blended.

If `displayType` is `linear`, the value in a patch cell is bit-shifted by the amount specified by the `shiftAmount` property and the result is used as the saturation of the shade of `color`. For example, if the value in a patch cell is 500 and `shiftAmount` is -3, `500 bitShift: -3` is 62. Therefore, the alpha blend factor is $62/255 = 0.24$. In this example, 76% of the Kedama World color and 24% of red are blended.

If `displayType` is `color`, the bit pattern for the value is interpreted as a pixel value of `R:G:B = 8:8:8` and just used to fill the corresponding pixel.

Why do we need these different display types? This is because the different commands exhibit different "characteristics". The `decayPatchVariable` and `diffusePatchVariable` commands in a ticking script give exponential rates of decline of cell values, as the new value is defined as a multiplication of the previous value. The `logScale` display type compensates such exponential behavior and gives smoother gradient of color. In typical simulations, `decay` and `diffuse` are the most commonly used so that one could say `logScale` display type is applicable for most common cases. The `linear` display type is convenient when the user uses `increase by` or `decrease by` assignment for the `patchValue` property. In that case, the rate of change in values tends to be linear, so `linear` display type gives smooth result. The `color` type is convenient when the values in cells are known to represent color pixel values.

Also, there are two sets of commands that extract or merge certain RGB color component from or to other specified patch variable. For example,

```
<receiverPatch><greenComponentFrom><otherPatch>
```

takes the cell values in `otherPatch` and merges the lower eight bit values of each cell into the green component position of each cells of `receiverPatch`. These commands effectively treat the values as color. In other words, the patch variables don't have the strong notion types of the cells, but the commands give the illusion of color type in patches.

The user visible characteristics of display type and display parameters are too close to their implementation details and color type handling is not sophisticated. Possible improvements are discussed in Chapter 8.

B.3 The Turtles

When we say “turtles” the word may imply two different entities. Of course, a turtle is a turtle. On the other hand, a breed of turtle that may contain a massive number of individual turtles, should be treated as a collective entity. The latter concept is very important in an interactive system, as the user should not be forced to interact with thousands of individual turtles.

In the viewer of an exemplar, there are three turtle specific categories; these are “kedama turtle”, “kedama turtle breed”, and “kedama turtle color”. Table B.1 shows the properties and commands available for the Kedama turtles. Again, “(r)” in Type column denotes that the property is read-only.

The “kedama turtle breed” category (in Figure B.1, it is shown as **Turtle breed properties**) provides the properties that is related to “a breed as one thing” aspect. In an analogy, the properties in the “kedama turtle breed” category are Smalltalk’s class side properties, while the properties in “kedama turtle” (other properties for turtles) are the instance side properties. One possible design could have separated viewers for the breed and turtles; however, in the current design, there is only one viewer for a breed of turtles, and the categorization in the viewer provides the distinction.

The `turtleCount` property in “kedama turtle breed” category controls the number of turtles in the breed. The modification of the value is immediately reflected and enough number of turtles are created or deleted. As you can see, the `turtleCount` is not associated with any particular turtle, but associated with the breed itself. There are some other experimental slots in the “kedama turtle breed” category, but these are only there for testing purposes.

The “kedama turtle” category has the properties and commands for turtles. The `x` and `y` properties represent the `x` and `y` coordinates. The readouts for these values in the viewer show the current value of the exemplar in the eToys World coordinates, but when the tiles for them are used in a script, they refer to the properties of individual turtles; i.e. each individual turtle has its own `x` and `y` property. Similarly, the `heading` represents the heading. Again, the readout for `heading` in the viewer shows the value of the exemplar in the eToys World coordinates.

The `color` and `visible` properties control the visual of turtles. Each turtle's `color` is used to fill the pixel at its (x,y) position in the Kedama World when the turtle is rendered, but it is done only when the `visible` slot is true.

The above mentioned properties, `x`, `y`, `heading`, `color`, and `visible` are “intrinsic”; in the sense that these values are owned by each individual turtle and the values themselves have the proper meanings. The other properties are non-intrinsic; they are defined in relationship with other objects.

The notable examples of non-intrinsic property for turtles are `angleTo` and `distanceTo` properties. The `angleTo` and `distanceTo` take an argument in turtle type and return the angle and distance to the argument, respectively.

Among the non-intrinsic properties, `patchValue` and `upHill` are defined in relation with a patch variable. `PatchValue` represents the value of the patch cell at which the turtle resides. It provides the way to access the patches from turtles. The `upHill`, which is a read only property, returns the direction toward the patch cell that holds the largest number among the cells around the turtle position. Both properties take an extra argument of patch variable type and calculate the value in the patch variable.

The getter of `turtleOf` takes an exemplar as an argument and returns a turtle in a breed at the argument's position. If there is no such turtle in the breed, `turtleOf` returns itself as the default value.

A turtle can clone itself. A special kind of property called `getReplicated` returns a clone when the property is accessed in a script. It is implemented as a property so that it can return a value.

A turtle also has some color related commands. In the “kedama turtle color” category of the viewer, there is a command that copies the color of all turtles in the breed to a specified patch variable as 24-bit RGB value. Also, there is a command that copies back the values in the patch cells to individual turtle's `color`. In the category, there are variations of `patchValue` that can handle the RGB color components separately. The `red-`, `green-` and `blueComponent` slots let the turtle access the specified component (the eight bits mask corresponds to R, G and B in the patch cell value), instead of the full 32 bit of patch values. Again, this is a way to provide the illusion to treat the patch cell values as colors.

Appendix C

The Syntax Transformation Rules

The syntax tree of a user script is transformed to another syntax tree to execute the statement properly in the parallel or the sequential mode. First the attribute evaluator calculates the attributes of nodes in the syntax tree, and then the tree is transformed in an attribute-directed manner.

The attribute evaluator specification for implementing the PSW model defines 23 attributes and 59 attribution rules. Most of the attributes are internally used to calculate others, but a few are carried to the next tree transformation stage. These attributes are as follows:

isStatement A `MessageNode` may represent a “statement” that is a top-level message in a script and may be subject to the transformation. The `isStatement` attribute is Boolean-type and attached to a `MessageNode` to indicate the node is a statement or not.

statementType The `statementType` attribute is Symbol-type that is attached to a `MessageNode`. Its value can be `parallel`, `sequential`, or `none`. A `parallel` statement is a statement that is executed in the parallel mode by default. Likewise, a `sequential` statement is a statement that is executed in the sequential mode. A `none` statement doesn’t contain any turtle breeds therefore it can be executed in the normal way.

messageType The `messageType` attribute can be either `condition` or `none`. A conditional statement is represented as a `MessageNode` in the parse tree (whose selector is `ifTrue:ifFalse:`). Such a `MessageNode` has `condition` as its `messageType`. Otherwise it is `none`.

`rewritelInfoIn`, `rewritelInfoOut` At the transformation stage, certain variables will be modified. The `rewritelInfoIn` and `rewritelInfoOut` attributes are attached to `VariableNodes` that represents variables and specify which variables should be rewritten to what.

These are also used for blocks and statements. At the transformation stage, some blocks need to have an extra argument. This attribute for a block specifies the name of the extra argument. For a statement, the name of primary breed is stored in one of the attribute (`rewritelInfoIn`).

Why are there two In and Out variants? As `rewritelInfos` provide the names to which some variables' names are mapped, they represent a kind of "environment". A node sometimes needs to have a way to access the "enclosing" environment, and passes the modified environment to its sub trees. At a node, the evaluator copies the value of `rewritelInfoIn` of the parent node to `rewritelInfoOut`, calculates a different environment for sub-trees if needed, and stores it to `rewritelInfoOut` of the node. (In the AG term, this is some sort of "threading" [76].)

After calculating these attributes in the parse tree, the syntax tree is transformed.

The examples of the transformations are shown in Section 7.2.4. The following provides semi-formal specifications of the rewriting rule.

In the figure, the left-hand side such as `stmt` represents the original parse tree node and the tuples in the parenthesis represent the attribute names and their values at the node. On the right hand side, the result of transformation is shown. The figure means that if the values of the attributes match with the values shown on the left-hand side, the transformation rule is applied. The transformation is done in bottom-up manner; the transformer traverses the parse tree in bottom-up manner, looks for the applicable rule for each node, and replaces the node if there is one. In other words, when the transformer is looking at a node, its sub-nodes are already transformed. To denote the sub-nodes that are already transformed, the prime ' is attached in the figure.

The nonterminals are shown in this `monospace` font, and the terminal symbols and the attribute values are shown in `sanserif` font. The attribute name is shown in *italic*. The syntax variable `primBreed` represents the variable that refers to the primary breed object, and `t1`, `t2`, etc. are temporary variable whose names are chosen so that there is not name conflict.

Some variables in scripts need to be renamed to introduce the indirection explained in Chapter 7. The renaming is also done in attribute-directed manner. And, because the conditional statements can be nested, the name

stmt	$(statementType = none,$	→	stmt '
	$isStatement = true)$		
<hr/>			
stmt	$(statementType = sequential,$		primBreed doSequentially: [:t1
	$isStatement = true,$		stmt ']
	$messageType = none,$	→	
	$rewriteInfoIn = primBreed,$		
	$rewriteInfoOut = t1)$		
<hr/>			
stmt	$(statementType = parallel,$		primBreed doCommand: [:t1
	$isStatement = true,$		stmt ']
	$messageType = none,$	→	
	$rewriteInfoIn = primBreed,$		
	$rewriteInfoOut = t1)$		
<hr/>			
cond	ifTrue: [trueStmts]		primBreed doCommand: [:t1
	ifFalse: [falseStmts]		t1
	$(statementType = parallel,$	→	test: cond'
	$isStatement = true,$		ifTrue: [:t2 trueStmts']
	$messageType = condition,$		ifFalse: [:t3 falseStmts']].
	$rewriteInfoIn = primBreed,$		
	$rewriteInfoOut = t1)$		
<hr/>			
cond	ifTrue: [trueStmts]		primBreed doSequentially: [:t1
	ifFalse: [falseStmts]		t1
	$(statementType = sequential,$	→	test: cond'
	$isStatement = true,$		ifTrue: [:t2 trueStmts']
	$messageType = condition,$		ifFalse: [:t3 falseStmt']]
	$rewriteInfoIn = primBreed,$		
	$rewriteInfoOut = t1)$		

Figure C.1: The transformation rules for statements.

to which a variable is renamed should refer to the argument of the inner-most block. For a `VariableNode`, the `rewriteInfoIn` attribute provides the name to rename to. The basic idea is that the variable that refers to the primary breed is changed each time when the flow of control enters a block. Note that for variables in a statement whose `statementType` is `none`, the `rewriteInfo` will be `nil`.

In summary, the transformation rules for a variable are shown in Figure C.2.

$$\frac{\text{var} \quad (rewriteInfoIn = nil)}{\text{var}} \rightarrow \text{var}$$

$$\text{var} \quad (rewriteInfoIn = x) \rightarrow x$$

Figure C.2: The transformation rules for variables.

For a block, the same `rewriteInfoIn` attribute is used to indicate if the transformation is necessary. If the attribute is not `nil`, the named variable is added to the block argument. In summary, the transformation rules for blocks are shown in Figure C.3.

$$\frac{[stmts] \quad (rewriteInfoIn = nil)}{[stmts]} \rightarrow [stmts']$$

$$[stmts] \quad (rewriteInfoIn = t1) \rightarrow [:t1 | stmts']$$

Figure C.3: The transformation rules for blocks.

For non-statement message sends, only the statements with `parallel` or `sequential` type get transformed by inserting another `MessageNode` whose selector is either `doCommand:` or `doSequentially:`, respectively. Also, if it is a conditional statement, the message selector is changed from `ifTrue:ifFalse:` to `test:ifTrue:ifFalse:`, the receiver object is replaced according to the `rewriteInfoIn`, and the original conditional expression (`cond`) is moved to the first argument position.

In summary, the transformation rules for message sends is shown in Figure C.4.

stmt <i>(rewriteInfoIn = nil)</i>	→	stmt'
cond ifTrue: trueBlock ifFalse: falseBlock <i>(rewriteInfoIn = var,</i> <i>messageType = condition)</i>	→	var test: cond' ifTrue: trueBlock' ifFalse: falseBlock'
stmt <i>(statementType = parallel,</i> <i>rewriteInfoIn = var,</i> <i>rewriteInfoOut = t1)</i>	→	var doCommand: [:t1 stmt']
stmt <i>(messageType = sequential,</i> <i>rewriteInfoIn = var,</i> <i>rewriteInfoOut = t1)</i>	→	var doSequentially: [:t1 stmt']

Figure C.4: the transformation rules for message sends.

Appendix D

The Execution Engine for Kedama

A set of primitives, or the execution engine, for optimizing Kedama's execution has been written. Most of the primitives take homogeneous arrays as arguments, and iterate over them to mutate some of the arrays. Also, there are primitives used to optimize the similar operation at a particular index in the arrays, or a particular turtle. Since the action on a particular turtle would involve lots of boxed floating point number computation, doing the computation in the C-compiled code gives us visible performance gain. One way to look at the Squeak VM equipped with Kedama primitives is that the virtual machine has the instructions for vectors, or the machine is a virtual vector machine for the Squeak environment. All the Kedama primitives are written in Squeak and translated to ANSI C.

In the following, the primitives are listed and explained.

D.1 Numeric Operations

There are 22 primitives for numeric operations, and they can be categorized in 4 (2x2) groups.

One of the two categorizations is whether it is an operation on "array and array" (array-array) or on "array and scalar" (array-scalar). Another categorization is whether it is arithmetic or comparison.

An array-array primitive applies a certain operator to the corresponding values in the two argument arrays, and stores the results into the corresponding slots in the third array:

type	name	type	name
array-array	primitiveAddArrays	array-scalar	primitiveAddScalar
	primitiveSubArrays		primitiveSubScalar
	primitiveMulArrays		primitiveMulScalar
	primitiveDivArrays		primitiveDivScalar
	primitiveRemArrays		primitiveRemScalar
	primitiveEQArrays		primitiveEQScalar
	primitiveNEArrays		primitiveNEScalar
	primitiveGTArrays		primitiveGTScalar
	primitiveGEArrays		primitiveGEScalar
	primitiveLTArrays		primitiveLTScalar
	primitiveLEArrays		primitiveLEScalar

Table D.1: Array Operations

```
1 to: size - 1 do: [:index |
  result at: index put: (array1 at: index)  $\oplus$  (array2 at: index)]
```

where the operation, \oplus , may be addition, subtraction, multiplication, division, or remainder for arithmetic, or for comparison, it may be equal, not-equal, greater, greater-or-equal, less-than, or less-or-equal. In Table D.1, these primitives' names are listed. As you can see, the middle part of primitive names stand for different operations.

An array-scalar primitive perform a certain computation on each value in the argument array with the scalar argument, and stores the results into the corresponding slots in the third array:

```
1 to: size - 1 do: [:index |
  result at: index put: (array1 at: index)  $\oplus$  scalar1]
```

When a primitive is invoked, the arguments (three arrays for array-array operations or two arrays for array-scalar operations) should have the same length; otherwise the primitive raises an exception. The types of `array1` and `array2` may be either `KedamaFloatArray` or `WordArray`. The type of the scalar argument for array-scalar primitives may be `Float` or `Integer`.

For an arithmetic primitive, the type of result array is either `WordArray` or `KedamaFloatArray`. If the both arguments are integral value or arrays of integral values, the result array should be a `WordArray`, otherwise, it should be a `KedamaFloatArray`. For a comparison primitive, the result array should be a `ByteArray`. If these type constraints don't agree, the primitive raises

type	name
array-array arithmetic	primitiveOrByteArray primitiveAndByteArray primitiveNotByteArray

Table D.2: Logical Operations

type	name
assignment with scalar	primitivePredicateAtAllPutBoolean primitivePredicateAtAllPutColor primitivePredicateAtAllPutNumber primitivePredicateAtAllPutObject
assignment with array	primitivePredicateReplaceBytes primitivePredicateReplaceWords

Table D.3: Predicated Array Assignment Operations

an error. For each slot of the `ByteArray`, 1 (that denotes true) or 0 (that denotes false) is stored.

To support these four variations of the input types, there are four distinct inner loops in the primitives (to accommodate the dynamic type on the statically-typed C arithmetic operators). The types are checked dynamically and one of these four loops is selected based on the types.

For the division and remainder, there may be a case where divided by zero occurs in the primitive. In this case, 'NaN' is stored in the resulting array.

D.2 Logical Operations

There are primitives for logical `and`, `or`, and `not` primitives. Unlike the numeric operations described above, these three logical operations “destructively” update a given argument array. (Perhaps this behavior is not preferable.)

The `and` and `or` primitives take two `ByteArrays`, and updates the first argument (the receiver in Squeak’s messaging semantics). These two `ByteArrays` should be have equal length. The `not` primitive “reverses” (i.e., make all 0s to 1s and vice versa) the slots.

type	name	type	name
turtles	primTurtlesForward	scalar	primScalarForward
	primSetHeading		primSetScalarHeading
	primGetHeading		primGetScalarHeading
	primTurtlesSetX		primScalarSetX
	primTurtlesSetY		primScalarSetY
	primGetAngleTo		primGetScalarAngleTo
	primGetDistanceTo		primGetScalarDistanceTo
	primGetPixels		primGetScalarPixel
	primSetPixels		primSetScalarPixel

Table D.4: Turtle Operations

D.3 Predicated Array Assignment

The predicated array access primitives is somewhat similar to the array assignment, but do it for selected slots in the array. A such primitive takes the destination array, a predicate `ByteArray` that specifies which slots should be mutated, and a source value of the assignment. There are variations for different types of the array and different “source” argument and also whether it is a scalar or vector. The variations are shown in Table D.3. Four `AtAllPut` variants takes a scalar value of `Boolean`, `Color`, `Float` or `Integer`, or generic object, and put them to the slots. Since the representation of the `Color` and `Boolean` are different in the homogeneous arrays and in Squeak image, the primitive converts the scalar value so that it can be stored into the destination homogeneous array.

The source value of `primitivePredicateReplaceBytes` and `primitivePredicateReplaceWords` primitives are arrays. Since the format in the source array is the same, there is only two variations for byte objects and word objects.

D.4 Primitives for Turtle Actions

The actions and properties that involve turtles and have primitive supported are `forward`, `turn by`, `x`, `y`, `heading`, `angleTo`, and `distanceTo`. (`turn by` can be written in terms of the setter of `heading`, so they are unified here.)

There are the array versions as well as the scalar versions for them. All of these takes relevant homogeneous arrays that represents the turtles. The primitives that modify some of the argument arrays honor the predicates array and only mutate the entries that are specified by the predicates. The

type	name
support function	degreesFromX:y: degreesToRadians: radiansToDegrees: scalarXAt:xArray:headingArray:value:destWidth: scalarYAt:yArray:headingArray:value:destHeight:

Table D.5: Support Functions for Turtle

type	name
other primitives	drawTurtles makeMask: makeMaskLog zoomBitmap
	makeTurtlesMap
	randomIntoFloatArray randomIntoIntegerArray randomRange kedamaSetRandomSeed

Table D.6: Other Primitives

scalar versions take the index as an argument, and execute the specified command only at the specified index (only if the corresponding predicate is true).

The last four in the Table D.4 deal with the patch variable. In the primitives, the coordinates of turtles is rounded, and the corresponding cell in the patch variable is accessed. Again, the predicates is honored.

There are some supporting functions that are not called from Squeak directly but from the primitives. Such functions are listed in Table D.5. These include the radian and degree conversion. They also converts the 0 direction. eToys deals with degrees with “north” as 0 direction, while the 0 direction for radians is “east”. The scalarX... and scalarY... are called from primTurtlesForward and primScalarForward.

D.5 Other Primitives

There are other primitives (shown in Table D.6). `drawTurtles` render the turtles as dots onto a `Form`. `makeMask` and `makeMaskLog` convert the patch variable's content into color values. `makeMask` does it with a linear function while `makeMaskLog` does it with a logarithmic function. These are called "Mask" because the resulting `Form` has alpha channels and blended onto the Kedama World. `makeTurtleMap` supports the `turtleAt` intrinsic property. When it is called, all turtles `who` value is stored in a "map". This map is then accessed by turtles to see if some other turtles resides at its position. There are a few random number related primitives. `randomIntoFloatArray` and `randomIntoIntegerArray` generate bulk of random numbers and store them into given array.

Excluding trivial initializations, there are 61 primitives and 7 supporting functions in the set of primitives.

Publications

Refereed Journal Paper

- Yoshiki Ohshima, Ken Wakita, Masataka Sassa: A Report on Porting the Programming Environment Squeak to Sharp Zaurus and its Evaluation, *Transactions of Information Processing Society of Japan: Programming*, vol. 41, no. SIG-9 (PRO 8), pp. 62–77, Nov. 2000 (in Japanese).

Refereed Conference Papers

- Yohei Ikezoe, Akira Sasaki, Yoshiki Ohshima, Ken Wakita, Masataka Sassa: Systematic Debugging of Attribute Grammars. *AADEBUG 2000*, pp. 235–240, 2000.
- Yoshiki Ohshima, John Maloney, Andy Ogden: The Parks PDA: A Handheld Device for Theme Park Guests in Squeak. *OOPSLA 2003 Practitioners Reports in OOPSLA Companion proceedings*, pp. 370–380, 2003.
- Yoshiki Ohshima, Kazuhiro Abe: The Design and Implementation of Multilingualized Squeak. *In proceedings of The First International Conference on Creating, Connecting and Collaborating through Computing 2003*, pp. 44–51, 2003.
- Kentaro Yoshimasa, Yoshiki Ohshima, Kim Rose: Developing Squeak-Based Curricula through a Collaborative "TIDE" Course at Kyoto University and UCLA. *In proceedings of The Second International Conference on Creating, Connecting and Collaborating through Computing 2004*, pp 152–159, 2004.

- Michael Rueger, Yoshiki Ohshima: TranSqueak - Making the World a Smaller Place On-the-Fly Translation of Etoy Projects and Instant Messaging. *In proceedings of The Second International Conference on Creating, Connecting and Collaborating through Computing 2004*, pp 110–116, 2004.
- Yoshiki Ohshima: The Early Examples of Kedama, A Massively Parallel System in Squeak. *In proceedings of The Third International Conference on Creating, Connecting and Collaborating through Computing 2005*, pp 93–100, 2005.
- David Smith, Andreas Raab, Yoshiki Ohshima, David P. Reed, Alan C. Kay: Filters and Tasks in Croquet. *In proceedings of The Third International Conference on Creating, Connecting and Collaborating through Computing 2005*, pp. 50–56, 2005.
- Yoshiki Ohshima: Kedama: A GUI-based Interactive Massively Parallel Particle Programming System, *IEEE Visual Language and Human Centric Computing 2005*, pp. 91–98, 2005.
- Yoshiki Ohshima: Semantics and Performance Considerations of Kedama, a GUI-based Massively Parallel Programming Language, *In proceedings of The Fourth International Conference on Creating, Connecting and Collaborating through Computing 2006 (to appear)*, 2006.

Reports and Workshop Papers

- Yoshiki Ohshima, Kazuhiro Kuroishi, Teruhisa Hirai, Masataka Sassa: A consideration on development and maintenance activity of a free software – a compiler generator Rie as an example, *Proceedings of 48th National Convention of Information Processing Society of Japan*, Vol. 5, pp. 143–144, 1994 (in Japanese).
- Yoshiki Ohshima, Ken Wakita: The Implementation of Concurrent Transactions (in Japanese), *SIGPRO Reseach Report of Information Processing Society of Japan*, No. 96-PRO-8, pp. 49–54, 1996.
- 大島芳樹, 脇田建, 佐々政孝: PDA 上で動作するオブジェクト指向仮想記憶の実現に向けて, *online proceedings of 2nd SPA workshop*, March 1999.

- 喜多 一, 高田 秀志, 吉正 健太郎, 上野山 智, 渡辺 正子, Alan Kay , Kim Rose, 大島 芳樹: 京都大学・UCLA を結んだ遠隔講義による創造性教育, 日本教育工学会第 20 回全国大会 , No. 02-2a934-4, pp. 301–302, Sep. 2004.
- Yoshiki Ohshima: Kedama: A Massively Parallel Particle System Implemented on a Dynamic Language. *Revival of Dynamic Languages Workshop co-located with OOPSLA 2004*, Oct. 2004.

Thesis

- 大島芳樹: コンパイラ生成系 Rie の保守作業およびそれに関する考察 (学士論文), 東京工業大学 理学部 情報科学科, 1994.
- 大島芳樹: 並行言語への Concurrent Transaction 機構の導入 (修士論文), 東京工業大学 大学院 情報理工学研究科 数理・計算科学専攻, 1996.

Books

- BJ・アレン コン (著), キム・ローズ (著), 大島 芳樹 (監修), 喜多 千草 (監訳), 片岡裕子 (訳), 高田秀志 (解説): 子どもの思考力を高める「スクイク」 - 理数力をみるみるあげる魔法の授業, WAVE 出版, Feb. 2005.

Articles

- 大島芳樹: PC UNIX プロジェクト第 8 回「PC に先進の OS 環境を NEXTSTEP 3.1J」, UNIX USER, No. 12, 1993.
- 大島芳樹: PC UNIX プロジェクト第 9 回「PC に先進の OS 環境を NEXTSTEP 3.1J(後編)」, UNIX USER, No. 1, 1994.
- 大島芳樹: PC UNIX プロジェクト第 12 回「Solaris 2.1 for x86 (前編)」, UNIX USER, No. 5, 1994.
- 大島芳樹: PC UNIX プロジェクト第 13 回「Solaris 2.1 for x86 (後編)」, UNIX USER, No. 6, 1994.
- 大島芳樹: PC UNIX プロジェクト第 17 回「BSD/386 バージョン 1.1」, UNIX USER, No. 10, 1994.
- 大島芳樹: PC UNIX プロジェクト第 17 回「BSD/386 バージョン 1.1 後編」, UNIX USER, No. 11, 1994.

- 榊隆, 大島芳樹: オブジェクト指向言語, *BSD Magazine*, No. 3, pp. 119–121, 2000.
- 大島芳樹: Squeak, *Lightweight Language Magazine*, ASCII 出版, ISBN: 4-7561-4441-1, pp. 63–66., 2004.

Bibliography

- [1] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, “Back to the Future – The Story of Squeak, A Practical Smalltalk Written in Itself,” in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1997, pp. 318–326.
- [2] M. Montessori, *The Montessori Method*. Schocken, 1988, a paperback edition. The original book was published in 1912.
- [3] J. Piaget and B. Inhelder, *La Psychologie de L’Enfant*, 1966, a Japanese translation is available.
- [4] J. Bruner, *The Process of Education*, 1961, a Japanese translation is available.
- [5] Reggio Children, *The Fountains*, 1995.
- [6] S. Papert, *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, 1980, (Second Edition 1993).
- [7] E. F. Codd, *Cellular Automata*. Academic Press, 1968.
- [8] M. Gardner, “Mathematical Games,” *Scientific American*, October 1970.
- [9] P. W. Anderson, “More Is Different,” *Science*, vol. 177, pp. 393–396, 1972.
- [10] A. Kay, K. Rose, D. Ingalls, T. Kaehler, J. Maloney, and S. Wallace, “Etoys & SimStories,” February 1997, ImagiLearning Internal Document.
- [11] B.J. Allen-Conn and K. Rose, *Powerful Ideas in the Classroom*. Viewpoints Research Institute, 2003.

- [12] D. G. Deck and J. Rodríguez, “Squeak in Spain as Part of the LinEx Project,” in *Proceedings of the International Conference on Creating, Connecting and Collaborating through Computing (C5)*, 2004, pp. 160–165.
- [13] S. Konomi and H. Karuno, “Initial Experiences of ALAN-K: An Advanced LeArning Network in Kyoto,” in *Proceedings of the Conference of Creating, Connecting and Collaborating through Computing (C5)*, 2003, pp. 96–103.
- [14] D. G. Deck, A. P. Galán, A. P. Navarro, F. T. Escobar, J. F. Díaz, M. P. Conejo, and V. R. Cuerpo, *Squeak: Un Mundo para Aprender (in Spanish)*, 2005.
- [15] T. Yamamoto, *Play With Squeak (in Japanese)*. Shoeisha, 2003.
- [16] M. Resnick, *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds (Complex Adaptive Systems)*. The MIT Press, 1994.
- [17] U. Wilensky, “NetLogo,” 1999, <http://ccl.northwestern.edu/netlogo/>.
- [18] D. Hillis, *The Connection Machine*. The MIT Press, 1985.
- [19] E. Klopfer and A. Begel, “StarLogo Under the Hood and in the Classroom,” in *Kybernetes: The International Journal of Systems and Cybernetics*, vol. 32, February 2003, pp. 15–37.
- [20] K. Kahn, “ToonTalk: An animated programming environment for children,” *Journal of Visual Languages and Computing*, vol. 7, no. 2, pp. 197–217, June 1996.
- [21] T. Fujise, T. Chikayama, K. Rokusawa, , and A. Nakase, “Klic: A portable implementation of kl1,” in *Fifth Generation Computing Systems (FGCS '94)*, December 1994, pp. 66–79.
- [22] S. Cho, M. Kai, A. Kawai, T. Hino, S. Maeshima, and K. Kakehi, “Nigari - a programming language and environment for the first stage, leading to java world,” *Transaction of Information Processing Society of Japan: Programming*, vol. 45, no. SIG-9 (PRO 22), pp. 25–46, July 2004.
- [23] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi, *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press, 2001.

- [24] A. Kay and T. Kaehler, personal communication.
- [25] D. C. Smith, A. Cypher, and J. Spohrer, “KidSim: programming agents without a programming language,” in *Communications of the ACM*, vol. 37, no. 7, July 1994, pp. 54–67.
- [26] D. C. Smith, A. Cypher, and L. Tesler, “Programming by example: novice programming comes of age,” in *Communications of the ACM*, vol. 43, no. 3, March 2000, pp. 75–81.
- [27] Y. Harada and R. Potter, “Fuzzy rewriting: soft program semantics for children,” in *In Proceedings of Human Centric Computing Languages and Environment*, October 2003, pp. 39–46.
- [28] H. Tanuma, H. Deguchi, and T. Shimizu, “SOARS Spot Oriented Agent Role Simulator Design and Implementation,” in *AESCS '04*. Springer-Verlag, 2005, pp. 49–56.
- [29] T. Kurata, <http://www.cs.dis.titech.ac.jp/~soars/member/top.html>.
- [30] R. Pausch, T. Burnette, A. Capeheart, M. Conway, D. Cosgrove, R. DeLine, J. Durbin, R. Gossweiler, S. Koga, and J. White, “Alice: Rapid Prototyping for Virtual Reality,” in *Computer Graphics and Applications*, vol. 15, May 1995, pp. 8–11.
- [31] S. Cooper, W. Dann, and R. Pausch, “Teaching objects-first in introductory computer science,” in *ACM SIGCSE Bulletin*, vol. 35, January 2003, pp. 191–195.
- [32] A. Repenning, “AgentSheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments,” Ph.D. dissertation, Colorado University, 1993.
- [33] H. Richardson, “High Performance Fortran: history, overview and current developments,” April 1996, Technical Report. TMC-261, Thinking Machines Corporation.
- [34] G. H. Barnes, R. M. Brown, M. Kato, D. Kuck, D. Slotnick, and R. Stokes, “The ILLIAC IV Computer,” *IEEE Transactions on Computer*, vol. 8, no. 17, pp. 746–757, 1968.
- [35] R. Millstein, “Control Structures in Illiac IV Fortran,” *Communications of the ACM*, vol. 16, no. 10, pp. 621–627, 1973.

- [36] C. Lasser and S. M. Omohundro, *The Essential Star-lisp Manual*. Thinking Machines Corporation, 1986.
- [37] A. Chien, *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. The MIT Press, 1993.
- [38] J. K. Lee and D. Gannon, “Object Oriented Parallel Programming: Experiments and Results,” in *Supercomputing*, 1991, pp. 273–282.
- [39] K. Taura, S. Matsuoka, and A. Yonezawa, “An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers,” in *Principles and Practice of Parallel Programming (PPOP)*, 1993, pp. 218–228.
- [40] T. Yanagawa and K. Suehiro, “Software System of the Earth Simulator,” *Parallel Computing*, vol. 30, no. 12, pp. 1315–1327, 2004.
- [41] M. Daniels, “Integrating Simulation Technologies With Swarm,” *Agent Simulation: Applications, Models and Tools*, 1999.
- [42] One Laptop per Child, “\$100 Laptop Computer,” <http://laptop.org>.
- [43] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1983.
- [44] A. Borning, “ThingLab – A Constraint-Oriented Simulation Laboratory,” Xerox PARC, Tech. Rep. SSL-79-3, 1979.
- [45] M. Wolczko, “Self includes: Smalltalk,” Presented at the Workshop on Prototype-Based Languages, ECOOP ‘ 96.
- [46] M. K. Dalheimer, *Programming with Qt, 2nd Edition*. O’Reilly & Associates, 2002.
- [47] R. Eckstein, M. Loy, and D. Wood, *Java Swing*. O’Reilly, 1998.
- [48] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-Oriented Programming,” in *European Conference on Object-Oriented Programming*. Springer-Verlag, 1997, pp. 220–242.
- [49] Intel Corporation, “Intel IA-64 Architecture Software Developer’s Manual,” 2000.

- [50] D. Knuth, “Semantics of Context-Free Languages,” in *Mathematical Systems Theory*, vol. 2, no. 2, 1968, pp. 127–145.
- [51] M. Sassa, “Rie and Jun: Towards the Generation of all Compiler Phases,” in *Lecture Notes in Computer Science on Third International Workshop on Compiler Construction (CC)*, vol. 477, 1991, pp. 56–70.
- [52] K. Kennedy and S. K. Warren, “Automatic Generation of Efficient Evaluators for Attribute Grammars,” in *Third ACM Symposium on Principles of Programming Languages (POPL)*, 1976, pp. 32–49.
- [53] Y. Shinoda and T. Katayama, “Object Oriented Extension of Attribute Grammars and Its Implementation Using Distributed Attribute Evaluation Algorithm,” in *Lecture Note in Computer Science on Workshop on Attribute Grammars and their Applications*, vol. 461, 1990, pp. 177–191.
- [54] T. Teitelbaum and R. Chapman, “Higher-order attribute grammars and editing environments,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990, pp. 197–208.
- [55] B. Silverman, personal communication.
- [56] A. Begel, personal communication.
- [57] Y. Ohshima and K. Abe, “The Design and Implementation of Multilingualized Squeak,” in *Proceedings of the Conference on Creating, Connecting and Collaborating through Computing (C5)*. IEEE, 2002, pp. 44–51.
- [58] Y. Ohshima, K. Wakita, and M. Sassa, “A Report on Porting the Programming Environment Squeak to SHARP Zaurus and its Evaluation (in Japanese),” *Transaction of Information Processing Society of Japan: Programming*, vol. 41, no. SIG-9 (PRO 8), pp. 62–77, November 2000.
- [59] The Unicode Consortium, *The Unicode Standard, Version 4.0*. Addison-Wesley, 2003.
- [60] K. Handa and A. Tanaka, “personal communication.”
- [61] Y. Ohshima, J. Maloney, and A. Ogden, “The Parks PDA: a Hand-held Device for Theme Park Guests in Squeak,” in *Practitioners Report on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2003, pp. 370–380.

- [62] D. A. Smith, A. C. Kay, A. Raab, and D. P. Reed, “Croquet - a collaboration system architecture.” in *C5*, 2003, pp. 2–9.
- [63] M. Tokoro, *Computational field model: Toward a new computing model/methodology for open distributed environment*, 1992, pp. 3–14.
- [64] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
- [65] D. Ingalls, *Design Principles Behind Smalltalk*. BYTE Magazine, August 1981.
- [66] C. Hewitt, P. Bishop, and R. Steiger, “A Universal Modular Actor Formalism for Artificial Intelligence,” in *International Joint Conference on Artificial Intelligence*, August 1973, pp. 235–245.
- [67] A. Kay, “The Early History of Smalltalk,” in *ACM SIGPLAN Conference on History of Programming Languages*, 1993, pp. 69–95.
- [68] —, personal communication.
- [69] I. P. Goldstein and D. G. Bobrow, “A Layered Approach to Software Design,” Xerox PARC, Tech. Rep. CSL-80-5, 1980.
- [70] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, “Traits: Composable Units of Behaviour,” in *European Conference on Object-Oriented Programming (ECOOP)*, 2003, pp. 248–274.
- [71] M. Guzdial and K. Rose, *Squeak: Open Personal Computing and Multimedia*. Prentice Hall, 2002, ch. 2: An Introduction to Morphic: The Squeak User Interface Framework, pp. 39–68.
- [72] J. H. Maloney and R. B. Smith, “Directness and Liveness in the Morphic User Interface Construction Environment,” in *ACM Symposium on User Interface and Software Technology (UIST)*, 1995, pp. 21–28.
- [73] D. Ungar, “Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm,” in *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, 1984, pp. 157–167.
- [74] H. Schorr and W. Waite, “An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures,” *Communications of the ACM*, vol. 10, no. 8, pp. 501–506, August 1967.

- [75] Y. Ohshima, “Squeak vm for ipaq and other windows ce platforms,”
<http://www.is.titech.ac.jp/~ohshima/squeak/WinCE/wince.html>.
- [76] M. Sassa, H. Ishizuka, and I. Nakata, “Rie, a Compiler Generator Based on a One-pass-type Attribute Grammar,” in *Software - Practice and Experience*, vol. 25, no. 3, March 1995, pp. 229–250.