Kedama: A GUI-based Interactive Massively Parallel Particle Programming System

Yoshiki Ohshima

VPRI Technical Report TR-2005-001

Viewpoints Research Institute, 1025 Westwood Blvd 2nd flr, Los Angeles, CA 90024   t: (310) 208-0524

Kedama:
A GUI-based Interactive Massively Parallel Particle Programming System

Yoshiki Ohshima
Twin Sun, Inc.
360 N. Sepulveda Blvd. Suite 1040
El Segundo, CA USA 90245
Email: yoshiki@squeakland.org

Abstract

Decentralized and complex systems can be powerful tools for modeling concepts in mathematics, science and engineering. StarLogo and NetLogo were the first systems to allow middle and high school children to build dynamic models from many thousands of programmable particles.

This has inspired Kedama: an authoring system that simplifies the scripting and construction of massively parallel models to allow more students to join in this new rich environment. This has given rise to new user interface and programming language designs.

Kedama has the following notable features: 1) Scripts in Kedama can be written in a graphical tile scripting interface. Its users are alleviated from syntax errors, and all program elements are visually presented to the user. 2) Kedama is an extension of Squeak eToys: children who have learned to "script in the large" can use same techniques to "script in the small" with many thousands of objects. 3) The semantics and syntax are simplified but retain full power of expression. 4) The UI in Kedama is dynamic: scripts and other properties in Kedama can be dynamically modified while scripts are kept running. 5) Kedama is more portable, being able to run on more than two dozen types of platforms, including the most used around the world. 6) Kedama is fast. Various simulations typically run about 3 times faster than previous systems. This improvement made it possible to write different classes of examples.

1. Introduction

Many physical, biological and social phenomena, as well as mathematical concepts can be modeled as decentralized and complex systems. The simulations of such systems can be written with massively parallel particles. In a typical decentralized system, the behavior of each particle tends to be very simple, yet the interaction between them or even just the sheer number of them produces interesting emergent behavior.

The combination of a simple program and complex behavior is attractive to educators who try to use new tools in classrooms especially in math and science. If students could construct their own simulations and explore the problem domain, they would reach a better understanding of the simulation and simulated phenomena.

To materialize the idea, Resnick proposed a system called StarLogo [1]. StarLogo, and Wilensky's NetLogo [2], are designed for high school students' use. These systems have been accepted by many educators and students.

1.1. The Challenges

If StarLogo and NetLogo are already popular, why is the author working on yet another system? This is because the author sees an opportunity to provide a more accessible, user-friendly system. Below is the list of areas that can be improved.

First, none of the existing systems, as far as the author's knowledge, doesn't allow direct manipulation of the simulation. For example, in StarLogo and NetLogo, there are separated 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 and the simulation is reverted to the initial state. To allow the users to freely explore the domain and do "what-if" simulations, it is desirable that the entire system is kept executing while the user is modifying the system.

Also, the syntax and semantics of the scripts could be much simpler and suitable for a particle system. The syntax of Logo [3], which StarLogo and NetLogo are based on, is adequate if the system has only one entity that the user interacts with. For example, if the user writes a setColor command in Logo, there is no ambiguity in regard to which object will change its color. However, once the system starts having other kinds of objects, the ambiguity must be resolved.

The last area to mention is the performance and implementation. The popular implementation language of this kind of system, Java, provides good performance in general cases but it is not suitable to implement highly-specialized, fine-tuned vector operations that are needed in massively parallel particle simulations. Also, the static nature of the Java language makes it harder to implement a flexible object system. Also, the latest Java implementation is available only on a limited set of platforms.

1.2. Contributions

Based on the observations in the previous section, the author has implemented a new massively parallel particle system called Kedama. Kedama has the following notable features.

Visual Scripting Kedama provides a visual programming interface for writing massively parallel particle simulations. The interface resembles the well-documented Squeak eToys so the user can learn it quickly.

Direct Manipulation The data and code are shown as graphical objects so that they can be manipulated directly.

Cleaner Syntax and Semantics The uniform object-oriented syntax is more suitable to write parallel particle programs. The semantics is straightforward and analogous to Squeak eToys.

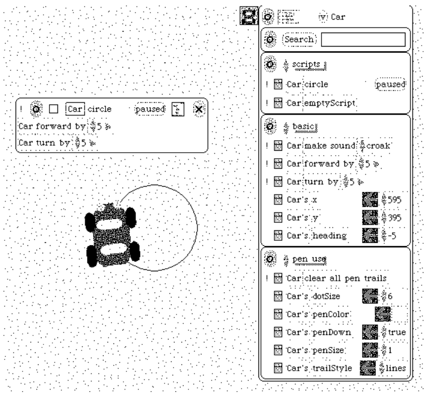
Performance Although it is hard to do fair comparisons, a simple equivalent program in Kedama typically runs 3 to 10 times faster than StarLogo and NetLogo. This performance improvement opens up the possibility to write new and qualitatively different classes of examples.

Portability Kedama is implemented on a highly-portable, reasonably-fast, open-source VM. The VM allows custom routines ("primitives") to be compiled by an optimizing C compiler to optimize the low-level operations in Kedama. The VM and primitives are so portable that it has ported to many uncommon platforms.

In short, Kedama shows that a simpler, more dynamic, user-friendly and fast implementation of massively parallel particle system is possible.

1.3. The Organization of the Paper

The following sections are organized as follows. In Section 2, the overview of the Squeak eToys system that



```
! Car circle          paused
Car forward by 5
Car turn by 5
```

```
Car
Search
scripts
! Car circle               paused
! Car emptyScript
basic
! Car make sound croak
! Car forward by 5
! Car turn by 5
  Car's x                  595
  Car's y                  395
  Car's heading            -5
pen use
! Car clear all pen trails
  Car's dotSize            6
  Car's penColor
  Car's penDown            true
  Car's penSize            1
  Car's trailStyle         lines
```

Figure 1. A screenshot from a typical eToys session

Kedama is based on is introduced. In Section 3, the language of Kedama is described. In Section 4, the semantics of scripts in Kedama is described. In Section 5, the implementation of Kedama is briefly explained. In Section 6, an example, performance evaluation, and the Kedama's multilingual feature are introduced. Finally, further related works that are not covered in Introduction are discussed, and the conclusion of the paper is given.

2. Squeak eToys: New Opportunity

Squeak eToys [4] [5] is a visual, tile-based scripting system designed primarily for fifth and sixth grade (11 and 12 years old) children. In Squeak eToys, a user can directly manipulate graphical objects on screen, and construct a program by using symbolic and graphical representations of objects. In the rest of the paper, Squeak eToys is referred to as "eToys".

EToys is drawing upon the past ideas, such as Bruner's constructivist theory of education and Papert's Logo. EToys is a tool that mimics a kind of toy so that the children can have fun to play with it. However, it is designed to let children build simulations and programs so that they can engage mathematical and scientific studies.

Since its public release in 1999, eToys has been enjoying its growing popularity around the world; there are schools officially using eToys in Spain [6], Japan [7], Germany, Canada and the US. Also, there are several books published about eToys in different languages ([5], [8]). Its popularity and wide acceptance led us to design a system that keeps the similarity to eToys.

2.1. Graphical Objects, Viewers and Scripts

An object in eToys is represented as a visible, interactive, graphical entity. Each object has properties or *slots*. Slots include intrinsic x and y coordinates as well as other non-intrinsic ones that are defined in terms of the relationship with other objects, such as isUnderMouse (set to true when the mouse cursor is pointing the object).

In a typical session, the user paints his own painting with eToys' paint tool, and then the system automatically converts the painting into a graphical object.

EToys provides a tool, called a *viewer*, to "look into" the graphical objects. A viewer provides the symbolic view of the associated object in the form of readouts of the slots. In Figure 1, there is an object that looks like a car, and the viewer of the car is shown on the right edge. The viewer also holds the tiles that represent the commands that the object understands. The execution typically causes some change of the properties. Let us call the dynamic effects caused by the execution of command *actions*.

The user can create and edit *scripts*. Editing a script includes adding and removing commands, dropping a slot onto the argument position of a command, changing the numerical arguments, and expanding the arithmetic expressions. Figure 1 shows a script with turn by 5 and forward by 5 commands.

A script can be marked as "ticking". A ticking script repetitively executes itself at a regular time interval. Multiple commands in a script are executed in a top-down manner. For example, the script in Figure 1 makes the car drive in a circle, because "it is moves a little and turns a little over and over again".

The first example in Logo was a turtle making a circle. In eToys, it is a car making a circle. However, the biggest difference is that scripts can be modified at any moment, even as they are running. Any modification such as adding a command or changing a parameter takes effect instantly. This dynamic nature of the system lets the children try different options and explore the problem domain quickly.

2.2. The Tile Scripting Language

The syntax of the tile language in eToys is object-oriented; namely, a command in eToys is defined in terms of the *receiver* object and the *message* sent to the receiver with some arguments.

All the commands in eToys take a form of:

<receiver><command><zero-or-more args>

Note that this form is *very* close to the Squeak's textual syntax. The semantics of this command is also very close to the Squeak's message passing paradigm. In fact, a tile script is converted to a textual method in a straight-forward way, and then, internally, the converted method is executed by the Squeak interpreter.

The language of eToys is statically-typed. When the user trys to combine tiles by dropping one onto another, it will succeed only if these tiles satisfy the type restriction. There is visual feedback that indicates the type conformance. The language is prototype-based. When the user add a variable or a script to an object, the individual object changes its "shape" and behavior.

2.3. Squeak Implementation

The implementation of eToys is also suitable to build a customized and optimized particle system. EToys is built on the top of a dynamic object-oriented language called Squeak [9]. Squeak offers an open-source and reasonably fast virtual machine (VM); the VM is ported to more than two dozens of platforms. Also, Squeak provides a portable way to write the customized VM routines called *primitives*. A primitive is typically written in Squeak itself and translated into ANSI compliant C code. The code can be compiled by an optimizing compiler, and linked together with the VM to improve the performance of certain operations. In this way, the primitives are reasonably well optimized, and highly portable among platforms.

The public version of Squeak only has a byte-code interpreter, but doesn't come with a JIT compiler. However, as shown in the section 6.2, Kedama spends most of the time in primitives so that the interpretation overhead is very small.

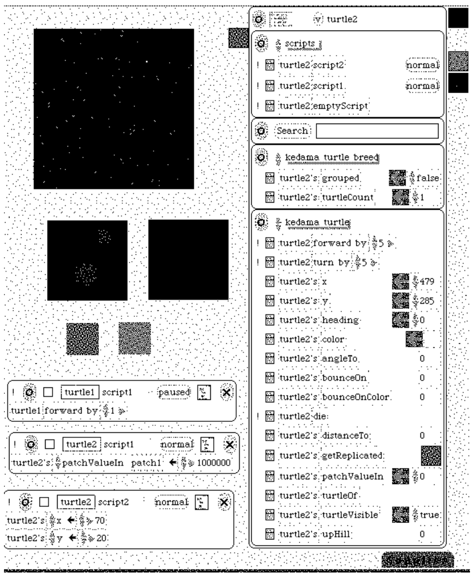
3. The Kedama System

In this section, the overview of Kedama system is described. The objects that consist of the system and the slots and the commands the objects have are also explained.

3.1. The Objects in Kedama

In addition to the objects of eToys, Kedama defines three new kinds of object. The most important one is the *parallel turtle*. A parallel turtle represents a group of homogeneous turtles, often explained as "a breed of turtle". One of the other objects is called a *Kedama world* that represents 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. 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.

Similar to the objects in eToys, these Kedama objects have graphical representations. The graphical representation of a breed of turtle is called an *exemplar*, and it is shown as a small square on the screen. A Kedama world is typically magnified and shown as a bigger square than its logical size. Turtles in the Kedama world are rendered as colored



```
turtle2
scripts
! turtle2 script2    normal
! turtle2 script1    normal
! turtle2 emptyScript
Search
kedama turtle breed
turtle2's grouped       false
turtle2's turtleCount   1
kedama turtle
! turtle2 forward by 5
! turtle2 turn by 5
turtle2's x             479
turtle2's y             285
turtle2's heading       0
turtle2's color
turtle2's angleTo       0
turtle2's bounceOn      0
turtle2's bounceOnColor 0
! turtle2 die
turtle2's distanceTo    0
turtle2's getReplicated
turtle2's patchValueIn  0
turtle2's turtleOf
turtle2's turtleVisible true
turtle2's upHill        0

! turtle1 script1   paused
turtle1 forward by 1

! turtle2 script1   normal
turtle2's patchValueIn patch1 ← 1000000

! turtle2 script2   normal
turtle2's x ← 70
turtle2's y ← 20
```

Figure 2. A screenshot of Kedama session

pixels. A patch variable is shown as a square without magnification. The value of each cell is converted to a color and the color is shown at the corresponding pixel. If specified, the cells are rendered in the corresponding Kedama world as well.

Figure 2 shows a screen shot of a typical session. The Kedama world is shown at the top-left, two patch variables are shown at the middle, and two exemplars are shown below.

The viewer for each different type of object provides different sets of properties and commands. The user combines the commands and properties from viewers of different objects to construct scripts.

3.2. The Trade off: Big vs. Small Feature Set

When one defines a programming system, there is a trade off on the size of the feature set the system provides. With more features, the user could write more complex programs. On the other hand, of course, too many features can easily confuse the user.

This observation applies more to Kedama. The primary audience of Kedama is school students, not necessarily a technical group. The system should not overwhelm the audience. Also, Kedama's programming interface is graphical. The icons, tiles, and buttons to control too many features would occupy more screen "real estate".

Based on this observation, the development of Kedama has been carried out by an incremental approach. It started from a minimum system, and once the bare minimum version started running, examples from other systems were tested on Kedama to determine the most important features. After some iteration, examples that are unique to Kedama were tested as well to refine the feature set.

The resulting feature set is interestingly concise. This is the direct consequence of the characteristics of decentralized systems; as described in Section 1, the description of behavior of objects in a decentralized and complex system tends to be very simple, yet the interaction between many objects creates non-trivial simulations. This means that the features to describe the behavior can be simple.

Table 1 lists the slots and commands of Kedama objects. It lists the names and types of slots. The slots marked as "(r)" are read-only, and others allow read and write.

3.3. Slots and Commands of Parallel Turtles

Parallel turtles have slots that can be classified into two different flavors; one is intrinsic and the other is non-intrinsic, or defined in relation with other objects.

Intrinsic ones, x, y, heading, color and visible, are defined in a straightforward way. Note that the position x and y) are defined in the Kedama world coordinates, however. Also note that the turtles don't "collide" with each other unless the user writes a script to detect the collision.

Among the non-intrinsic slots, patchValue and upHill are defined in the relationship with a patch variable. The getter of patchValue gets a patch variable argument, and returns the cell value at which the turtle resides. The setter of patchValue, similarly, gets a patch variable and a number as arguments and stores the specified number to the patch cell at the turtle's position. The upHill slot, which is read only, gets a patch variable as an argument, and returns the direction of steepest gradient in the patch from the position of the turtle.

The getter of grabAtMyPositionOf gets 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, grabAtMyPositionOf returns itself. A special kind of getter of getReplicated creates the receiver's clone and returns the newly created clone. The read-only slots, angleTo and distanceTo return the angle and distance to the specified turtle, respectively.

The commands of the parallel turtle are simple. The semantics of forward by and turn by is the same as eToys'. The die command removes the executing turtle from the system.

The turtleCount slot is special in the sense that it controls the property of a breed itself. The change of the turtleCount value is instantly reflected and the number of turtles in the breed changes.

Table 1. The slots and commands of Kedama

Slot or command name	Type
Turtle slots (intrinsic)	
x, y, heading	Number
visible	Boolean
color	Color
Turtle slots (derived)	
patchValue	Number (r)
upHill, angleTo, distanceTo	Number
getReplicated, grabAtMyPositionOf	Turtle (r)
Turtle commands	
forward by, turn by, die	
Turtle breed slots	
turtleCount	Number
Kedama world slots	
color	Color
pixelPerPatch	Number
topEdgeMode, bottomEdgeMode, leftEdgeMode, rightEdgeMode	Symbol
patchDisplayList, turtleDisplayList	Collection
Patch Variable slots	
color	Color
displayType	Symbol
shiftAmount, scaleMax, sniffRange, evapolationRate, diffusionRate	Number
Patch Variable commands	
decay, diffuse	

3.4. Slots and Commands of Kedama World

A Kedama world offers the following intrinsic slots: There are four "edge modes" that specify the default behavior for turtles. If a turtle is about to move off an edge of the Kedama world, the turtle's new position is determined by the corresponding edge's mode. If the mode is wrap, the turtle wraps around to the opposite edge. If the mode is bounce, the turtle bounces back. If the mode is stick, the turtle stops at the edge.

The color slot specifies the background color and pixelPerPatch specifies the magnification upon rendering.

A Kedama world keeps track of associated patches and breeds of turtle to be rendered on top of it. The slots, patchDisplayList and turtleDisplayList provide them respectively.

3.5. Slots and Commands of Patch Variable

A patch variable offers three different functions to map the integral values in its cells to colors. A slot called displayType specifies the function. The options are logScale, linear, and color. If displayType is logScale, the logarithm of the value is calculated first, and the log value is used to determine the saturation of the hue specified by the color slot. If displayType is linear, the value is shifted by the amount specified by the shiftAmount slot and the result is used as the saturation of the shade of color. If displayType is color, the value is interpreted as a pixel value of R:G:B = 8:8:8.

The decay command reduces all values at the rate specified by evapolationRate slot. Similarly, the diffuse command reduces all values, but it uses the average of the neighboring cells for the new value of a cell.

3.6. Plotting

In a complex system, it is important to extract a representative value of the entire system and plot a graph to understand the collective behavior. In a typical programming environment, the plotting feature is built-in so the user only needs to specify the variable to plot the graph for the variable.

Kedama, of course, allows the user to make a graph plotting. However, the biggest difference of Kedama is that the plotter object is no different from normal eToys objects; i.e., the plotter object is also tile-scriptable. The user can change the parameter of plotting in the same manner he manipulates other objects. The idea is to expose the otherwise hidden "under the hood" machinery to the user so that he can explore the system without requiring additional knowledge.

3.7. Color Manipulation

In Kedama, there is another set of features to support the bulk manipulation of colors of turtles. If a Kedama world can be "filled" with turtles; i.e. create a turtle at each grid point, whole new classes of examples are possible. One interesting class of examples is to have the turtles exchange their colors between themselves and patches and "wear" the color. Or, a kind of patch programming, where each cell is actually a turtle, can be possible, too. Thanks to the performance of Kedama, it is practical to do "fill the world" examples and create a real-time animation.

4. Semantics of Parallel Execution

In Kedama, the semantics of scripts needs some consideration because of the presence of parallelism. The parallelism should not only be accommodated, but in a way that is understandable to non-technical users without sacrificing the performance.

There are two important questions to be answered to define the semantics. These questions are “how many times is a command executed?” and “in what order are the commands executed?” The questions are answered by the key concept I call *line-wise synchronization* and *single thread execution*.

Suppose we have multiple commands in a script and one of them is a forward by command for a breed of turtle. Upon the execution, all the turtles in this breed move forward. In the other words, this command is executed a number of times that is equal to the number of the turtles. The *line-wise synchronization* concept means that a command is executed only after all the actions associated with previous commands are completed. In this example, all the turtles in the breed move forward before the next command in the script takes effect.

Another concept is the *single thread execution*; that is, for each invocation of a script, there is only one thread running through the entire script. If a command in the script doesn’t have any side-effect, its actions can be optimized. However, if an action caused by a command on a turtle has some side-effect, the side-effect is visible to the action on the next turtle. A Test-Yes-No command, which corresponds to an if-then-else statement in a typical language, is treated as a larger but otherwise normal command; i.e. for each turtle in the breed, the Test and execution of the commands in either the Yes or No clause are repeated.

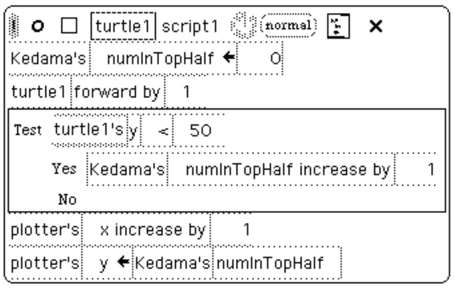
Figure 3 and Figure 4 depict the execution semantics. Figure 3 shows the sample script, and the black line in Figure 4 shows the flow of control.

The flow of control enters from the top of the script. The first command in the script resets the numInTopHalf slot of the object named Kedama to zero. Since this slot is defined for a non-turtle object, the assignment action is executed only once.

Then, the process threads through the forward by command. This is a turtle command thus executed for each turtle in the turtle1 breed. The jagged line in Figure 4 depicts that the action is taken as many times as the number of the turtles in the breed.

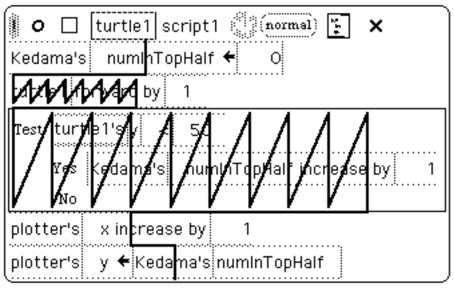
After the completion of the forward by actions on turtle1’ turtles, it executes the Test-Yes-No tile, which is treated as one “fat” command. Since it has a reference to the turtle1, this fat command is executed for every turtle in the breed in a serialized manner. For each turtle in the turtle1 breed, the command checks the turtle’s y coordinate and increase the numInTopHalf slot by one if y is less than 50. As a net effect, the value of numInTopHalf will indicate the number of turtles in the turtle1 breed whose y coordinate is less than 50.

Lastly, the thread updates the plotter’s x and y. These



```
turtle1 script1 normal
Kedama's numInTopHalf ← 0
turtle1 forward by 1
Test turtle1's y < 50
  Yes Kedama's numInTopHalf increase by 1
  No
plotter's x increase by 1
plotter's y ← Kedama's numInTopHalf
```

Figure 3. A sample script



```
turtle1 script1 normal
Kedama's numInTopHalf ← 0
turtle1 forward by 1
Test turtle1's y < 50
  Yes Kedama's numInTopHalf increase by 1
  No
plotter's x increase by 1
plotter's y ← Kedama's numInTopHalf
```

Figure 4. The thread of execution on the script

two lines are non-turtle command so they are executed just once.

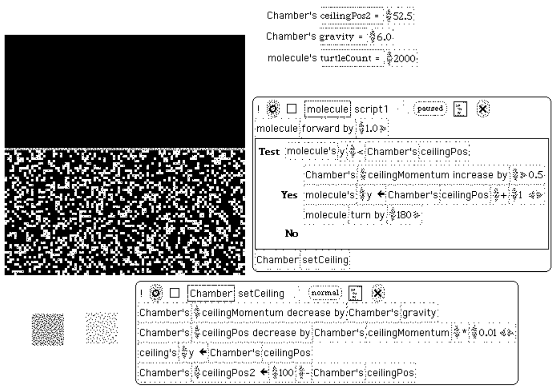
5. The Implementation

The implementation of Kedama can be divided into three layered parts. The upper level that the user interacts with is the user interface (UI) part. At the middle level, there is the internal representation of objects. At the bottom is the Kedama execution engine that contains the optimized primitives. All of these are implemented in Squeak. The line count of code is about six thousand.

Kedama’s UI extends the eToys’, which is written in a GUI framework of Squeak called Morphic [10]. In Morphic, all graphical entities, including viewers, tiles, and user objects are a sub-instances of the Morph class in the framework.

A careful design of the data representation of a parallel turtle is crucial to gain performance. In the current implementation of Kedama, the values of the same slot for all the turtles in a breed are stored in a homogeneous array. For example, the x coordinate of turtles in a breed are stored in an array of 32-bit floating point value, so are y, heading, and other user-defined slots. These arrays are legitimate Squeak objects and also have compatible memory format with arrays in C language. These arrays are then passed to the execution engine and manipulated uniformly.

Kedama implements a parse-tree-to-parse-tree transfor-



```
Chamber's ceilingPos2 = 52.5
Chamber's gravity = 6.0
molecule's turtleCount = 2000

! molecule script1 paused
molecule forward by 1.0
Test molecule's y < Chamber's ceilingPos
Yes  Chamber's ceilingMomentum increase by 0.5
     molecule's y ← Chamber's ceilingPos + 1
     molecule turn by 180
No
Chamber setCeiling

! Chamber setCeiling normal
Chamber's ceilingMomentum decrease by Chamber's gravity
Chamber's ceilingPos decrease by Chamber's ceilingMomentum * 0.01
ceiling's y ← Chamber's ceilingPos
Chamber's ceilingPos2 ← 100 - Chamber's ceilingPos
```

Figure 5. A gas tank with a moving ceiling

mation so that the commands to the parallel turtles are translated differently. During this transformation, tiles are identified as a parallel command or not, and added the invocations of the execution engine primitives and special control structure in it.

The Kedama execution engine is a set of primitives. The execution engine contains 38 exported primitives and 7 supporting functions. Some of the primitives are used to optimize the parallel commands such as forward by and turn by. Others support the rendering such as the patch variable color mapping. The engine also contains various vector arithmetic primitives. These are used to improve the performance of "vector and scalar" or "vector and vector" operations that show up in user-defined expressions.

The graphical tile scripting system limits the available operations and the combinations of types; however, this is a blessing rather than a curse. The limiting nature of possible operations makes it possible to provide primitives for these operations beforehand.

6. Kedama in Action

Kedama is and is not a prototype system. While it will evolve over the time, it is stable, has a good feature set, and ready to use in classrooms. In this section, an example that takes advantage of Kedama's expressive power and performance, some performance evaluation, and the multilingual feature to put it in classrooms around the world are discussed. To see more examples, refer to another paper by the author [11]. Also, the common examples presented in [1], such as the ant colony and termites, are successfully replicated in Kedama.

6.1. A Gas Tank Example

Figure 5 is a screen shot from a simulation of gas particles in a chamber. There is a moving ceiling represented by another breed of turtle, and there is also a gravity field. The ceiling is pulled downwards by the gravity field thus trying to do a constant acceleration motion. However, when a particle hits the ceiling, it gives a small momentum upwards to the ceiling and bounces back. The edge mode of bottom, left, and right are set to bounce so as the ceiling closes to the bottom edge, the more particles hit the ceiling and give more momentum to the ceiling. With thousands of particles, individual behavior is averaged out and the ceiling position reaches equilibrium.

The user can change the parameters such as the number of particles and the gravity constant dynamically. When the user does so, the animated motion from the old to new equilibrium is compelling for the user to see why the ideal gas equation stands.

The code for this example is simple. There are only ten lines in a ticking script, and about seven in the scripts to set the initial condition.

6.2. Performance

The data layout in homogeneous arrays is in the native C format so that they can be manipulated directly by C-compiled primitives. If the Squeak VM spends a large potion of the execution time in primitives, we can say that the byte-code interpretation is not the bottle neck; other implementation languages wouldn't give too much performance improvement over current Kedama's implementation. In simple examples such as "Bouncing Atoms" in [11], the system spends about 80% of time in primitives of Kedama. On a Pentium-M 900MHz computer, "Bouncing Atoms" with 20,000 particles runs at about 32 frames per second. For comparison, an equivalent example in NetLogo 2.1 runs about 1.8 fps with "Exact Turtle Positions & Sizes" setting turned on (default) or 8.7 fps with off. Note that the latter is closer to what Kedama does, but still Kedama outperforms more than 3 times.

As the model becomes more complicated, the advantage of having the pre-compiled parallel primitives becomes smaller. However, even in the worst case, where all operations are serialized, Kedama retains the performance advantage.

6.3. Localization

Kedama should be accessible to the students all over the world. Since the primary audience of Kedama is school students, non-English speaking students should be able to use the system in their native language.

The author and his colleagues have added the multilingualization feature to Squeak [12] so that it can handle multi-byte character sets. Also, Squeak has a simple message catalog translation mechanism called "Babel", implemented by Diego Gomez Deck. With these mechanisms, and again, thanks to the limiting nature of tile-scripting, Kedama provides a way to translate *all* messages, tiles, and other UI related phrases visible to the user. The Japanese version is fully done, and the author expects that adding more languages is straight forward.

7. Related Work

As mentioned many times in this paper, StarLogo and NetLogo have been used in schools. Kedama is inspired by them, but the author believes that it overcomes some of the predecessors' drawbacks.

There are more advanced particle systems such as Swarm [13] and Agentsheets [14]. Swarm is an Objective-C library and provides a powerful simulation environment, but it isn't a dynamic manipulation environment, and requires far more complex textual coding to do simple programs.

Agentsheets supports a wider variety of functions and has a GUI tile scripting interface. However, it doesn't support full direct manipulation, such as dynamic code editing.

8. Conclusion

A massively parallel particle system called Kedama is presented. Kedama has graphical tile scripting that is aimed to non-technical students. Kedama's dynamic interaction lets the user change the code and parameters while the system is running and freely explore the problem domain.

The implementation utilizes a highly-portable, open-source virtual machine. The performance is typically faster than the similar systems implemented in Java.

The use of tile scripting has many advantages. Not only does it make the system more accessible to end users, the execution engine can provide optimized primitives.

Kedama is now stable enough to try in actual classrooms. Colleagues and I are planning to use the system at high schools this year (2005). The package is available for everyone for free via the Internet at `http://www.is.titech.ac.jp/~ohshima/squeak/kedama/`, or an Internet search with "Kedama Squeak".

There are some areas to be improved. For example, the integration with Point type adds the ability to use point- and vector-functions in expressions will add another level of expressiveness.

References

[1] M. Resnick, *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds (Complex Adaptive Systems)*. MIT Press, 1994.

[2] U. Wilensky, "NetLogo," 1999, http://ccl. northwestern.edu/netlogo/.

[3] S. Papert, *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, 1980, (Second Edition 1993).

[4] A. Kay, K. Rose, D. Ingalls, T. Kaehler, J. Maloney, and S. Wallace, "Etoys & SimStories," February 1997, ImagiLearning Internal Document.

[5] B.J. Allen-Conn and K. Rose, *Powerful Ideas in the Classroom*. Viewpoints Research Institute, 2003.

[6] D. G. Deck and J. L. R. Rodriguez, "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.

[7] 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 Collabora ting through Computing (C5)*, 2003, pp. 96–103.

[8] T. Yamamoto, *Play With Squeak (in Japanese)*. Shoeisha, 2003.

[9] 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*, 1997, pp. 318–326.

[10] M. Guzdial and K. Rose, *Squeak: Open Personal Computing and Multimedia*. Prentice Hall, 2002, ch. An Introduction to Morphic: The Squeak User Interface Framework, pp. 39–68.

[11] Y. Ohshima, "The Early Examples of Kedama, A Massively Parallel System in Squeak," in *Proceedings of the Conference of Creating, Connecting and Collabora ting through Computing (C5)*, 2005.

[12] 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.

[13] M. Daniels, "Integrating Simulation Technologies With Swarm," *Agent Simulation: Applications, Models and Tools*, 1999.

[14] A. Repenning, "Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments," Ph.D. dissertation, Colorado University, 1993.