Here, There and Everywhere – On the Recurring Use of Turtle Graphics in CS1

Michael E. Caspersen and Henrik Bærbak Christensen
Centre for Experimental Systems Development
Department of Computer Science, University of Aarhus
DK–8200 Aarhus N, Denmark
{mec,hbc}@daimi.au.dk

Abstract

The Logo programming language implements a virtual drawing machine—the turtle machine. The turtle machine is well-known for giving students an intuitive understanding of fundamental procedural programming principles. In this paper we present our experiences with resurrecting the Logo turtle in a new object-oriented way and using it in an introductory object-oriented programming course. While, at the outset, we wanted to achieve the same qualities as the original turtle (understanding of state, control flow, instructions) we realized that the concept of turtles is well suited for teaching a whole range of fundamental principles. We have successfully used turtles to give students an intuitive understanding of central object-oriented concepts and principles such as object, class, message passing, behaviour, object identification, subclasses and inheritance; an intuitive understanding of recursion; and to show students the use of abstraction in practice as the turtles at a late stage in the course becomes a handy graphics library used in a context otherwise unrelated to the turtles.

1 Introduction

It is our firm conviction that the primary aim for an introductory programming course is that students learn fundamental programming principles and techniques. The mastery of a programming language is, of course, necessary, but we view it as a secondary concern; we want to focus on fundamental principles and general techniques as early as possible and thereafter unfold these throughout the course.

Contrary to this, most introductory programming texts focus on the programming language, often described in a bottom-up fashion starting with the simpler constructs of the language and progressing to more advanced constructs. Only subordinate to the presentation of the language *constructs* follows the presentation of programming *techniques*; however, all too often these programming techniques are not even explicit in textbooks.

Another motivation for our approach is that most people learn more easily *through the concrete towards the abstract* [5, 9]. Having seen constructs and techniques being applied in an appealing intuitive way, and thereafter mimicking these to solve similar problems, like in a *craft's apprenticeship*, provides an excellent basis for a later thorough and more abstract treatment. In this way the students have a practical experience to ground the abstract treatment.

1.1 The Inverted Curriculum

Our view is not a novel one as is evident from many papers from past SIGCSE conferences [4, 6, 7, 11, 12]. Bertrand Meyer [8] coined the term "the inverted curriculum" (or "consumer-to-producer-strategy") meaning that important topics and concepts should be covered first by *using* classes solely through their abstract specifications, and only then the students learn about the internals of classes. A simplified variant of Meyer's vision is the objects-first approach which is prevailing in many new textbooks, but still many of these books are structured on the basis of the constructs in the programming language and not on the basis of the language independent concepts, principles and techniques that the students are supposed to master by the end of the course.

Of course, in order to be able to focus on programming techniques and apply these in concrete programs, it is necessary to be—at least to some extent—fluent in a programming language. However, we do not want the learning of the language to take over and become the primary concern, especially not in the beginning of the course. What we want is to *jump start* the students so

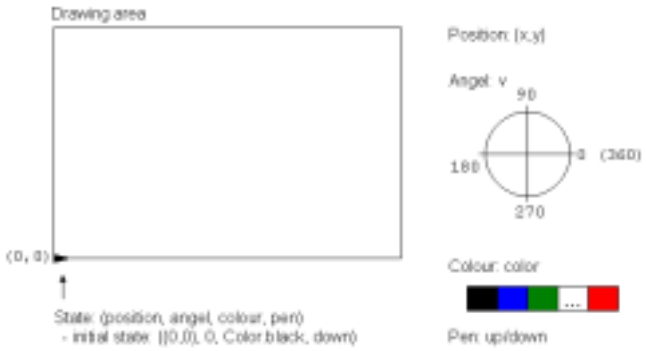

Figure 1: Architecture of the Turtle Machine

that they, as early as possible, can start writing interesting and challenging programs based on the fundamental principles and techniques that are our primary concern in the course: programs as physical models, objects, behaviour, classes, state, control flow, parameterisation, design by contract (specifications), inheritance, etc.

In order to facilitate a jump start of CS1, we have developed a Java package, TURTLES, that takes as it starting point the familiar turtle graphics developed by Seymour Papert and others at MIT in 1967 [10, 1]. We use it to give an intuitive introduction to concepts such as state, control flow, and parameterisation. Somewhat to our surprise, it turned out that the TURTLES package could play many more roles within CS1 than initially anticipated: It has become a recurring vehicle for introducing such diverse topics as objects and classes, object models, recursion, polymorphism and class hierarchies. Indeed, turtles popped up here, there, and everywhere...

In the current version of our introductory programming course we are using the programming language Java which is also the language of choice for the presentation in this paper.

2 The Turtle Machine

The original Logo turtle machine is a virtual drawing machine that uses the metaphor of a turtle with a coloured pen moving around in a Cartesian drawing area to produce drawings. The state of the turtle machine can be described as a 4-tuple: a turtle position (x,y)-coordinates, an angle, a colour and a up/down status for the pen. Initially the turtle is placed in the lower left corner (0, 0), the angle is zero, the colour is black and the pen is down (figure 1).

The set of instructions for the machine is minimal; only nine instructions are used to operate the machine (see Table 1).

Table 1: Instruction set for Turtle Machine

Command	Behaviour
`move(l)`	move l units in current direction
`moveTo(x, y)`	move to position (x, y)
`turn(d)`	increase the angle d degrees
`turnTo(d)`	set the angle to d degrees
`center()`	move to center
`penUp()`	lift the pen
`penDown()`	lower the pen
`setColor(c)`	set the pen's colour
`clear()`	clean the drawing area

3 The Turtle Machine Resurrected: Turtles

The original turtle machine sprang out of the *procedural programming paradigm* that views a program as a sequence of instructions carried out by some virtual machine. In contrast the *object-oriented programming paradigm* views a program as a model where model elements are objects that have behaviour and interact with other objects. Thus—in our object-oriented CS1 course—the turtle *machine* has naturally been replaced by turtle *objects*. In our Java implementation, there is no machine that executes turtle commands, instead there are objects that exhibit turtle behaviour; behaviour that is described by the Turtle class. The instruction set in Table 1 is replaced by (otherwise semantically equivalent) methods in the Turtle class.

This change of view and paradigm comes natural because the original metaphor of a turtle moving around on a drawing area is inherently an object-oriented model.

4 Jump Starting

At the beginning of the course we teach the concepts from the concrete towards the abstract. We start by introducing our "mascot" turtle with the odd, but short, name `t`. `t` lives in a sandbox (the large drawing area) and has a pen that leaves a trail when it moves around. `t` has *behaviour*: move-, turn-, and pen-behaviour. `t` exhibits the move-, turn-, and pen-behaviour when we pass it the message to do so, e.g. `t.move(100)` tells `t` to move 100 units forward. Before we show a computerised turtle, we actually let the audience command the lecturer around the floor in an attempt to produce a rectangle—while it reinforces the intuitive understanding of the behaviour concept, it also 'breaks the ice' between audience and lecturer as the audience for a short period is 'in control of the lecturer' as they pass messages: "Henrik, please move 2 meters" and so on. Controlling the turtle (or lecturer) also brings an intuitive understanding of the importance of the sequencing of messages passed, the control-flow. Parameterisation

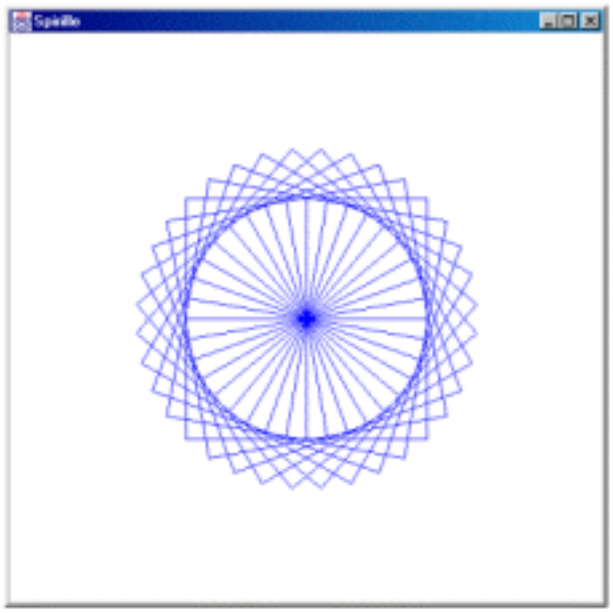

Figure 2: The "Spirille"

Program 1 The "Spirille" program

```
// 36 squares each turned an angle
// of 10 degrees from the previous
public class Spirille {
  public static void main() {
    Turtle t = new Turtle();

    t.setColor(Color.blue);
    t.center();
    for (int i = 0; i<36; i++) {
      for (int j = 0; j<4; j++) {
        t.move(100);
        t.turn(90);
      }
      t.turn(10);
    }
  }
}
```

also follows naturally as e.g. the 'move'-behaviour needs additional detail, namely the actual distance to travel.

The computerised turtle is then described through online viewing, editing, and running of Java code using a laptop computer connected to a projector.

We motivate loops in control flow in order to avoid textual repetition, e.g. looping four times over `{t.move(100); t.turn(90);}` is easier than writing eight turtle messages. This quickly leads to quite interesting drawings as illustrated in figure 2 that is produced by program 1.

At this point, through a concrete and highly visual metaphor, students have already an intuitive first understanding of fundamental object-oriented concepts: object, object identification and message passing, as well as fundamental procedural concepts: state, flow of control (including loops) and parameters. The immediate visual feedback from the program makes it easy for students to identify logical programming errors and helps the inexperienced student; at the same time the material is still advanced enough to challenge those students that are already familiar with the basic topics.

The lab exercises are about making simple drawing (a flag and a house), nested drawings (pyramid seen from the top, a high-rise block, etc.) and animations (various objects that move around).

Typically, students can be divided into two groups; one group of students tend to use the relative commands `turn` and `move` whereas others are more comfortable with the absolute commands `turnTo` and `moveTo`. We discuss the different approaches in class, and in particular we investigate the difference of using the relative and the absolute commands. This turns into a discussion on important and fundamental software engineering issues such as generality, modifiability and reusability of programs.

5 Objects and Classes

A natural next step is to introduce *two* turtles into the same drawing area. This seemingly trivial addition is actually an intuitive and powerful way to introduce the students to another important range of fundamental concepts in object orientation—a trivial and natural step in an object-oriented language but difficult in the original turtle machine.

Having two turtles makes the importance of *object identification* clear: How else can you identify the actual turtle to whom a message is sent? Another reinforced point is that the two turtles have different states though they share a common behaviour—they appear and draw in different areas of the drawing area. From this example it is natural to discuss the benefits of categorising objects with common behaviour, and give examples from everyday life where we classify concepts and phenomena. Introducing the notion of a (Java) class is thus relatively easy.

6 Class Hierarchies and Procedural Abstraction

The next step is to introduce procedural abstraction through defining new methods to draw, say, a rectangle. At first sight this seems like an overwhelming task to do in the second lecture as the only way to add a new method in Java is either to introduce it into the Turtle implementation or to extend the Turtle class and introduce the method in the subclass. The first alternative is not an option—primarily because the turtle is provided

Program 2 Procedural abstraction and parameterisation

```
class SkilledTurtle extends Turtle {

  void rectangle(int w, int h) {
    t.move(w); t.turn(90);
    t.move(h); t.turn(90);
    t.move(w); t.turn(90);
    t.move(h); t.turn(90);
  }

  public static void main() {
    SkilledTurtle t = new SkilledTurtle();

    ... t.rectangle(100, 50); ...
  }
}
```

as a Java package and secondly because we do not want to expose the implementation with all its details of the Java graphics. But the second option, to extend the Turtle class, turns out to be quite natural as described below.

6.1 Class Hierarchies

What do you do when you want your turtle to learn new "tricks", say, drawing a rectangle? You train your turtle until its behaviour extends to include the ability to draw rectangles—and your turtle becomes a *skilled* turtle.

By focusing on the idea of 'extending behaviour' the Java syntax for declaring subclasses seems feasible (program 2 and 3). We show the students how (program 2), and they are able to mimic the idea in exercises where turtles with new special skills are required as exemplified in program 3. We do not dwell on abstract, complex, properties of inheritance and class hierarchies; rather, we show how this technique—grounded in an intuitive understanding of "training turtles"—can be used to solve a concrete problem. In this way we have an excellent basis for a thorough treatment later in the course when the students have concrete experience and an intuitive understanding of inheritance. Also, the students have seen an aspect of what inheritance is actually used for—and in the end we find this is the basic purpose of the course: not merely to understand language constructs and object oriented principles but being able to apply them to solve recurring problems in computer science.

6.2 Procedural Abstraction and Design by Contract

Based on the metaphor of skilled turtles the focus is turned to the problem of "training". The first skilled turtle is one that can draw rectangles, and clearly, one wants to be able to define once and for all how to draw

Program 3 Specialisations of turtles

```
class GeometryTurtle extends Turtle {
  void rectangle(int w, int h) { ... }
  void circle(int r) { ... }
  ...
}

class ArchitectTurtle extends Turtle {
  void window(int w, int h) { ... }
  void door(int w, int h) { ... }
  void roof(int w, int h) { ... }
  ...
}
```

a rectangle with width w and height h (program 2).

From the SkilledTurtle example (or similar ones) we initiate a discussion on the necessity of the last `t.turn(90)` in the procedure of program 2. The statement is superfluous as far as the resulting drawing is concerned, but there are obvious reasons to include the statement: to leave the turtle in the same state as before the call, making it easier to make composite drawings by multiple calls (like the Spirille). The students understand the point, and hopefully valuable seeds have been sown.

On the basis of simple examples like this we discuss important fundamental principles such as design, specifications and the distinction of what and how. In the context of the turtles, it comes natural for the students to express sound and well established principles for procedural abstractions, and later in the course when things get more complicated, we return to this common ground and recall the principles.

The moral of the discussion is that we need to be precise about what we want a piece of software to do. The best way to express such requirements is by writing a functional specification; hence we introduce the notion of *design by contract* [8], and from then on we use the technique throughout the course. This is reinforced as we provide the specification of the Turtle as JavaDoc API documentation, thereby forcing the students to become acquainted with the standard way of documenting Java classes and packages.

7 Recursion and Fractals

A traditional way to introduce recursion is to compute factorials. We find this unfortunate, because it introduces the technique on a problem for which it is inefficient and an iterative solution is straight-forward to express. Contrary to this, we introduce recursion for problems where the recursive solution is effective and iterative solutions are difficult to express elegantly.

The students are asked to write a program that can produce the list of drawings, Triangle, Penta and Poly,

Figure 3: Triangle, Penta, and Poly

Program 4 Java code for triangle and penta

```
public void triangle(int l) {
  for (int i= 0; i<3; i++) {
    move(l);
    turn(120);
  }
}
public void penta(int l) {
  triangle(l/2);
    move(l/2);
  triangle(l/2);
    turn(120); move(l/2); turn(-120);
  triangle(l/2);
    turn(-120); move(l/2); turn(120);
}
```

in figure 3 (and the next seven figures which are given equally odd names). However, first we demonstrate how to write methods for the first two drawings (program 4).

As expected, the students produce eight new methods by copy-paste-and-substitute of the penta method. It works, but of course the students get the hunch that this cannot be the proper way to do it.

Once more we emphasise the notion of parameterisation, and we introduce the term `superTriangle(n)` to mean "a superTriangle of degree n". Defining `superTriangle(0)` to denote Triangle, `superTriangle(1)` to denote Penta and so forth, brings us more than half way towards the general solution; realizing that `superTriangle(-1)` does not make sense and handling this special case brings us the rest of the way (program 5).

The derivation is fairly easy; with little guidance the derivation is almost exclusively done by the students.

Program 5 A general (recursive) solution

```
public void superTriangle(int n, int l) {
  if ( n == 0 )
    triangle(l)
  else {
    superTriangle(n-1, l/2);
      move(l/2);
    superTriangle(n-1, l/2);
      turn(120); move(l/2); turn(-120);
    superTriangle(n-1, l/2);
      turn(-120); move(l/2); turn(120);
  }
}
```

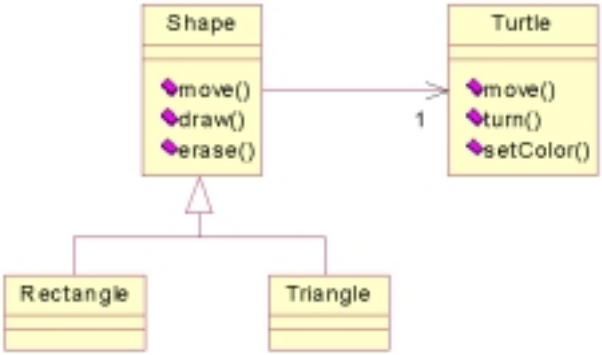

Figure 4: A hierarchy of geometric shapes

But even more interesting: Nobody mentions the notion of recursion; the solution just turns out to be what we call recursive.

8 Turtles as a Class Library

Later in the course, when we are covering more advanced object-oriented topics such as class hierarchies, polymorphism and application frameworks, we dig out the "old" TURTLES package and use it as just another class library. We also find it important for students to use class libraries and the accompanying documentation as early as possible in the undergraduate curriculum, as pointed out in e.g. [13].

8.1 Class Hierarchies and Polymorphism

We use geometric shapes as example of a class hierarchy. An abstract class `Shape` has concrete methods `move` and `erase` and an abstract method `draw` that is implemented in subclasses of the `Shape` class. Each `Shape` instance has a turtle associated that it delegates the drawing tasks to; in this way the turtle becomes our graphical drawing library effectively encapsulating the Java specific graphical toolbox (figure 4).

There is another important point in (re-)using the TURTLES package as a drawing toolbox: Abstraction is the key concept in programming, and the code which is the intense focus of design, development, and testing today (the implementation view, *how*), will be taken for granted next month and simply used (the specification view, *what*). In a similar vein, the turtle was "the problem" in the beginning of the course—now it is the solution to the problem of drawing shapes in a new and different context.

8.2 Application Frameworks

Before introducing the students to GUI-programming with AWT or Swing, we give a lesson about frameworks in general, and we exemplify by providing a sim-

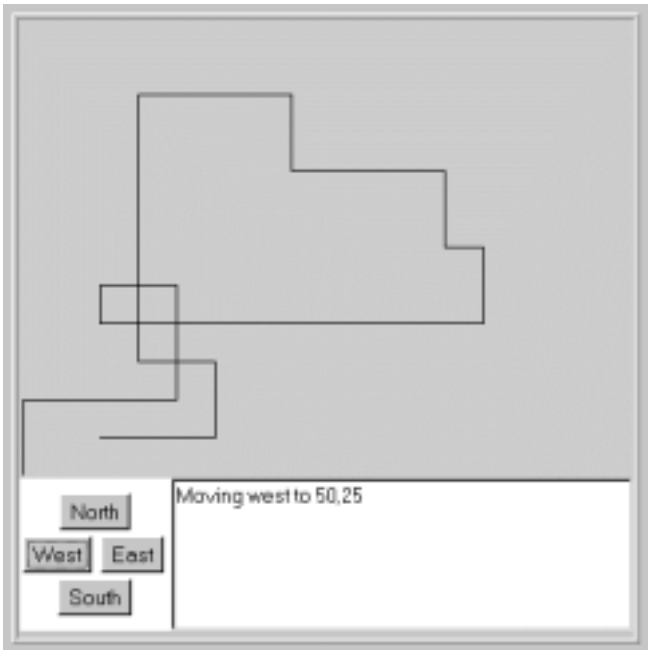

Figure 5: A turtle in the *Presenter* framework

ple framework for the students. The purpose of the framework, called *Presenter*, is to allow fast development of graphical presentations of a set of images (actually graphical components) and text, where the ordering in the set is arranged using a familiar navigational metaphor: The compass with directions north, east, south, and west.

Our initial instantiation of the framework is a multimedia presentation of the tomb of Tutankhamon—using the compass buttons the user can move between the different chambers of the tomb, each chamber described both in text and by a picture from the original opening of the tomb.

In an exercise the students are asked to program a turtle controller i.e. the buttons North, West, South and East must control the movement of the turtle (moving at right angles), as shown in figure 5. While it shows the turtle in yet another context the main point here is that the turtle's drawing area is actually a subclass of `java.awt.Component`, the basic graphical component in Java, and therefore the framework accepts to display the turtle drawing area. This way another important property of inheritance is demonstrated to the students; not as much as a language construct, but as a technique for solving a specific set of problems.

9 Conclusion

We have described our use of a Java package, TURTLES, which is an object-oriented variant of the classical turtle machine. Early in the course we are using the TURTLES package to jump start our CS1 course by giving an intuitive introduction to classical procedural concepts in the spirit of the Logo language, introducing only the most necessary constructs and only by example; we do not want to provide detailed explanations that will not be understood nor remembered at this early stage.

TURTLES is a great way to introduce simple as well as more advanced object-oriented concepts such as state, behaviour, object identification, inheritance, and polymorphism because the metaphor of a turtle on a drawing area is inherently an object-oriented model.

Furthermore, the TURTLES package has been successfully used to illustrate abstraction at a later stage in the course: while the semantics and details of turtles were the focus and problems in the early part of the course, it is simply used as a drawing class library in the later part of the course.

Though we have not conducted qualitative nor quantitative analysis of the effectiveness of our use of turtles to introduce object-oriented concepts to students, we have many indications of the positive effect. Our teaching assistants report that most students are proficient in basic object-oriented and procedural techniques early in the course, and students report using the turtles as fun and motivational. After all, this is not too bad.

Acknowledgements

We acknowledge Jens Bennedsen for stimulating discussions and collaboration during early development and use of the Turtle Machine. We also acknowledge Erik Martino for implementing the TURTLES package in Java.

References

[1] Abelson, H., and diSessa, H. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics.* The MIT Press, 1980.

[2] ACM SIGCSE. *The Papers of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education* (March 1993), vol. 25 of *SIGCSE Bulletin.*

[3] ACM SIGCSE. *The Papers of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education* (March 1995), vol. 27 of *SIGCSE Bulletin.*

[4] Astrachan, O., and Reed, D. The Applied Apprenticeship Approach to CS1. In *The Papers of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education* [3].

[5] Brightman, H. J. On Learning Styles. Tech. rep., Georgia State University, 1998. www.gsu.edu/~dschjb/masterteacher.html.

[6] Decker, R., and Hirshfield, S. Top-Down Teaching: Object-Oriented Programming in CS 1. In *The Papers of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education* [2].

[7] Hilburn, T. B. A Top-Down Approach to Teaching an Introductory Computer Science Course. In *The Papers of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education* [2].

[8] Meyer, B. *Object-Oriented Software Construction (2nd edition)*. Prentice-Hall, 1997.

[9] Myers, I. B., and McCaulley, M. *Manual: A Guide to the Development and Use of the Myers-Briggs Type Indicator*. Consulting Psychologist Press, 1985.

[10] Papert, S. *Children, Computers, and Powerful Ideas*. Harvester Press, 1980.

[11] Patti, R. E. The 'Procedures Early' Approach in CS 1: A Heresy. In *The Papers of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education* [2], pp. 122–126.

[12] Reek, M. A Top-Down Approach to Teaching Programming. In *The Papers of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education* [3], pp. 6–9.

[13] Tewari, R., and Gitlin, D. On Object-Oriented Libraries in the Undergraduate Curriculum: Importance and Effectiveness. In *The Papers of the Twenty-fifth SIGCSE Technical Symposium on Computer Science Education* (March 1994), vol. 26 of *SIGCSE Bulletin*, ACM SIGCSE, pp. 319–323.