ACTIONSCRIPT: A GENTLE INTRODUCTION TO PROGRAMMING

Stewart Crawford & Elizabeth Boese
Colorado State University
Fort Collins, CO 80523
(970) 215-6203 & (970) 491-7016
sgcraw@cs.colostate.edu & boese@cs.colostate.edu

ABSTRACT

In this paper, we propose the use of ActionScript as an introductory programming language. ActionScript has a Java-like syntax that allows the novice student to focus on basic language constructs and algorithms, without the need to delve into the enigmatic scaffolding and API code inherent with Java or C++, for example. The graphical nature of object creation also allows immediate and concrete examples that support an object-first pedagogy. The ActionScript environment is a natural for creating visually engaging animations and interactive projects. This engaging and less-confounding gentler approach to programming fires up student interest that in turn can help in the recruitment and retention of computer science students. Two example exercises are illustrated and their Java applet and ActionScript implementations are compared and contrasted. A framework placing ActionScript in curricula for CS specifically and the sciences more generally is discussed.

INTRODUCTION

For a novice coming to their first programming class, there is an aura of mystery in just getting their first program running. Within the context of the dominant instructional programming languages, Java and C++, there are scaffolding constructs to get past and often many syntactic details to specify, such as for object specification. How can the expression `public static void main` be given any meaningful explanation to a novice?

Our motivations, when introducing students to the fundamentals of computer programming and algorithmic problem solving, are (1) to structure student programming assignments so that they produce highly visual results and (2) to minimize the language overhead necessary to get to those results. Interactive projects and graphical animations provide a rich learning environment: the program s results are immediately visible which makes debugging easier and the final results more tangible and rewarding. We have found students eager to share these projects with friends and family, while we have never heard of a student eager to demo the traversal of a doubly-linked list to their mom and dad.

This engaging, gentler approach will positively impact CS recruitment and retention, currently major issues for CS departments across the nation. ActionScript enables students to develop sophisticated programs with graphical appeal with less scaffolding and frustration than traditional languages such as Java or C++. Not only does this help students over the initial barriers of a first course in programming, but it could also make the subject more enticing to diverse groups such as women, who traditionally have been underrepresented in CS enrollments.

Some alternatives have been proposed for minimizing language overhead. Developing applets provides one avenue to such graphical projects and reduces some of the overhead in getting graphical Java programs running. There is still a significant amount of language overhead to overcome in such assignments, though. Cooper, et al. [1], present a similar case for Alice, a 3D interactive animation environment for the teaching of introductory programming concepts. Their experience supports the idea that highly visual projects help students better understand the underlying state of a program and thus develop a better sense of program operation, in addition to the visual rewards of such work. However, some students have complained that such specialized teaching environments leave them without a real-world programming language upon graduation. David Reed [6] proposes using Javascript with HTML pages for programming projects, a solidly real-world environment. Students can start off quickly within the set of objects in the web s domain object model. This environment however is limited to the domain of web page objects and is not easily extensible to a wide range of other intriguing graphical student projects and provides only limited exposure to more advanced programming constructs.

Nell Dale [2] writes,

> *I do not believe that a consensus language for introductory computer science is possible. To some the choice of a language is almost a religious issue We must remember that a computer language is a vehicle, not an end in itself. Perhaps we can live in peace provided that we recognize that schools are as different as the students themselves We must tailor our curricula to the needs and expectations of our students within the context of giving them the best computing education that we can.*

The Final Report of the CC2001 committee [4] acknowledges that no ideal CS introductory sequence has emerged, and encourages continued innovation and experimentation in the introductory CS sequence. Given that more than 90% of introductory CS classes are taught in Java or C++ [3], in this paper we propose and present a kinder, gentler introduction to programming with ActionScript.

As a simple example of our approach, consider a program that makes a circle move across a frame on-screen. When written as a Java applet, there's essentially one line of code that makes the ball move, and all the other lines are overhead, i.e., code required to generate an applet in its web browser context and code required to draw a simple ball. In contrast, in the ActionScript solution, there is no programming overhead, since the object creation is done graphically with a drawing tool and the window creation is transparent.

There is only one line of code to write, and the student can immediately see its effect when they test the program. The results, right or wrong, are both immediate and visual, which strongly drives home the programming lesson without mystery.

The ActionScript language is an object-oriented Java look-alike, so the language syntax and semantics are full-strength, which follows all the way up through common object-oriented concepts. What is not necessary is early introduction of language constructs beyond the comprehension of introductory students. The immediate visual feedback strongly reinforces student learning. And students who go no further in programming classes will also then have a development tool at their command which has immediate job-market applicability.

In the following section, this paper explores two introductory lessons and contrasts implementations written as Applets and ActionScripts. After that, a framework placing ActionScript in curricula for CS specifically and in the sciences more generally is discussed.

EXAMPLES

The following examples were constructed to illustrate some of the salient differences between Java and ActionScript implementations and the required conceptual constructs that must be presented to an introductory student. We attempted to bring both the Java and ActionScript code into rough stylistic alignment for comparison purposes only and no style recommendation should be inferred; stylistic issues such as placement of braces will always be either a somewhat personal choice or dictated by project standards.

The lines of code are annotated in column one to reflect the basic concepts:

T – code for stepping through **T**ime
A – code for the basic **A**lgorithm
G – code for the **G**raphical object
S – the **S**caffolding code, stuff that just has to be there to make it work

Opening and closing braces, the basic syntax of a block structured language, are common to both environments and would be addressed identically, thus they make no net difference to this comparison and will be ignored in discussion.

Example One: the Rolling Ball

This first example is a simple exercise is to get an animation on the screen: a simple geometric object (a circle in this case) moves across the screen, and reverses direction when it meets the left or right edge of the window. A simplest introductory example leaves out the conditional statement(s) necessary to change direction. However, adding the conditional makes the moving ball persistent on-screen, and introduces conditionals as a first algorithmic programming construct.

The concepts necessary for this example are:

1. creation of a graphical object;
2. a repeated time step over which an animation will occur;
3. object motion between time-steps and a reversal of direction at a boundary.

Example One: Code for the Rolling Ball

JAVA:

```
S  import java.awt.*;
S  import java.applet.*;
S  public class BallBounce extends Applet
     {
G       int     _x = 0;
G       int     _y = 50;
G       int     side = 20;
A       int     step = 10;
S       public void paint(Graphics g)
        {
T         while (true)  // loop forever
          {
G           g.clearRect(_x, _y, side, side);
A           if ( _x + side > 440  ||  _x < 0 )  { step = -step; }
A           _x = _x + step;
G           g.fillOval(_x, _y, side, side);
T           try { Thread.sleep( 10 );} catch (Exception e) {}
          }
        }
      }
```

ACTIONSCRIPT:

In a new movie clip, the student draws a circle; then they drag it from the library onto the stage and name it ball ; then they select the ball with a mouse-click and attach this code onto it.

```
T     onClipEvent(enterFrame)  // do this once every movie frame
      {
A           if ( _x < 0  ||  _x > 440 ) { step = -step; }
A           _x = _x + step;
      }
```

In Frame #1, this initialization code is added:

```
A      ball.step = 10;
```

Several points can be immediately noted in this comparison.

No graphics code (G) is in the ActionScript implementation, since graphical objects are created with drawing tools. This is a conceptual win, having graphical objects created graphically. This overcomes the reliance on mathematics to define objects, so

there need be no discussion of diameters or bounding boxes in order to create a circle. Additionally, more complex objects can be created quickly and easily with drawing tools rather than mathematics, and they can be colored or shaded and placed on-screen with immediate visual feedback.

No scaffolding code (S) is in the ActionScript implementation. There is no code that just has to be there to make it work. The mysteries of an `import` statement, or of the construct `public class BallBounce extends Applet` need not confound the novice. The Java scaffolding is already somewhat reduced because it is within the framework of an applet, and thus the code necessary to generate and draw in a window is already hidden from the student. Of course those Java statements can be explained away, as they have been explained to generations of students to date. But they still have a mystery, and when students do them wrong, things just don t work and the student does not know why. What helpful error message is generated when an import statement is omitted? In the four ActionScript statements, the timing step will require a bit of explanation, but the other three are directly approachable. Modify any of those three in a syntactically correct way, and the effect of the modification is immediately visible in program execution.

There will be some issues to control with ActionScript. ActionScript variables are untyped; they take the type of the thing most recently assigned to them. This will be a contentious issue, since strong type-checking is an axiom of modern software engineering. The decision for introductory CS, however, is when and how to introduce it to the student; this then is a matter of pedagogical style. Another issue that begins to appear here is that the code is in two different places: initialization on the main timeline and behavior on the object. It is easy in ActionScript to spread the code around to many places and then lose track of where it is; this needs to be constrained when presented to the students to avoid such confusion.

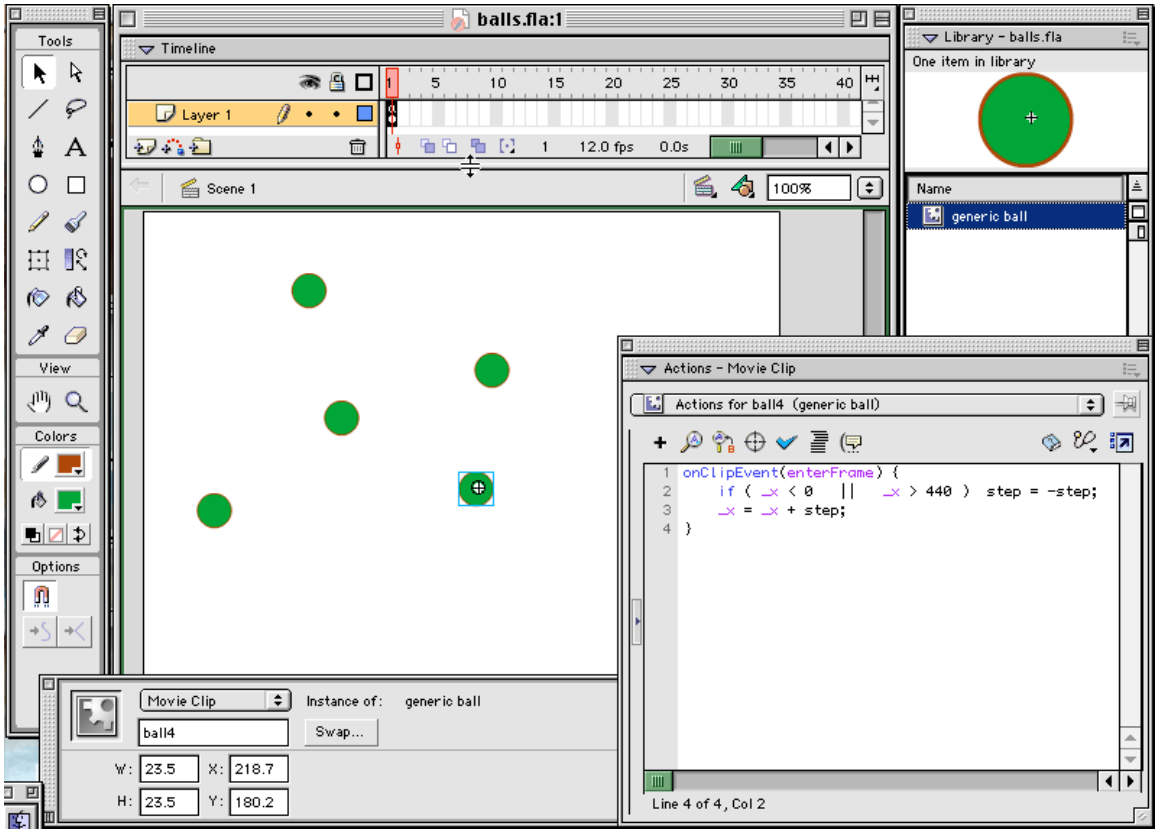
An object framework is already established. A point not immediately obvious is the implicit objectness of the ActionScript implementation. The `ball.step` construct already has the essence of an object and its property. This ActionScript exercise can be simply extended by: (1) copying the initially defined ball and pasting several repeatedly on the stage, (2) giving them each unique names, and then (3) setting up the initialization code in frame #1 to be something like this:

```
ball_1.step = 10;
ball_2.step = 5;
ball_3.step = 17;
ball_4.step = -3;
```

This is an easy lead-in to a concrete first implementation of Objects. In the library (see Figure 1) is the definition of a generic object. By dragging an object from the library onto the stage, the user creates an instantiation of that object in the world, as one would do with the `new` construct in an OO language. Each instantiation is its own separate world, witnessed by each having a `step` variable, with each `step` different. If the library object is opened and changed, all the balls change simultaneously. This is a simple and concrete introduction to objects, properties, and instantiations, in support of an object-first pedagogy.

In the applet example, there is no sense of an object. To make five balls onscreen, the procedural code would simply be replicated five times. Certainly the solution could be re-constructed to create a ball class and instantiate it several times with different initial conditions. That, however, would be a formidable amount of material to digest for a simple first assignment in programming, which this exercise represents.

Figure 1: Flash s ActionScript environment, showing the implementation of the multi-object version of Example 1. Note the library of generic objects in the upper left, and the one ball object has been instantiated several times in the main stage For the one selected ball (lower-right on the main stage), its basic properties are shown below and the program which animates it is shown in the Actions window on the lower right.



Example Two: I/O and Array Processing

ActionScript has an implicit time-stepping built-in, and animations such as Example 1 give an edge to ActionScript because of its timeline-based functionality. This second example removes that edge, presenting a standard user I/O and array processing exercise. The user enters a query term, then the program looks-up the term in an array of possibilities and outputs the corresponding response (or an error message if not found).

The algorithmic concepts central to this example are:

1. definition of arrays for query and response;
2. text input of query string;
3. event handling of completed query;
4. lookup array index of query and corresponding response;
5. text output of response.

For the sake of simplicity, the arrays are pre-defined and coded at initialization.

Example Two: code for I/O and Array Processing

JAVA:

```
S  import javax.swing.*;
S  import java.awt.*;
S  import java.awt.event.*;

S  public class query extends JApplet implements ActionListener  {
G   JTextField wordInField = new JTextField(10);
G   JTextArea defOut = new JTextArea(4,30);
G   JButton button = new JButton( "Look Up...");
A   String words[] = { "dog", "cat", "gerbil" };
A   String defs[] = {
A      "big fluffy playmate, eats bones, takes owner for walk",
A      "usually fluffy, chases birds, howls like a baby when in heat",
A      "likes to run up your sleeves when you take them out of cage"
        };
S   public void init( )  {
G      defOut.setLineWrap(true);
S      button.addActionListener( this );
S      setLayout(new FlowLayout());
G      getContentPane().add( wordInField);
G      getContentPane().add( button);
G      getContentPane().add( defOut);
    }
T   public void actionPerformed( ActionEvent ae )  {
A      String wordIn = wordInField.getText();
A      int i=0;
A      string def = "";
A      while( i<words.length && def.equals("") )  {
A        if( words[i].equals(wordIn) )  {
A            def = defs[i];
A            defOut.setText(defs[i]);
         }
A        i++;
       }
A     if( i==words.length)
A        defOut.setText("Sorry!\nCouldn't find that pet...");
    }
   }
```

ActionScript:

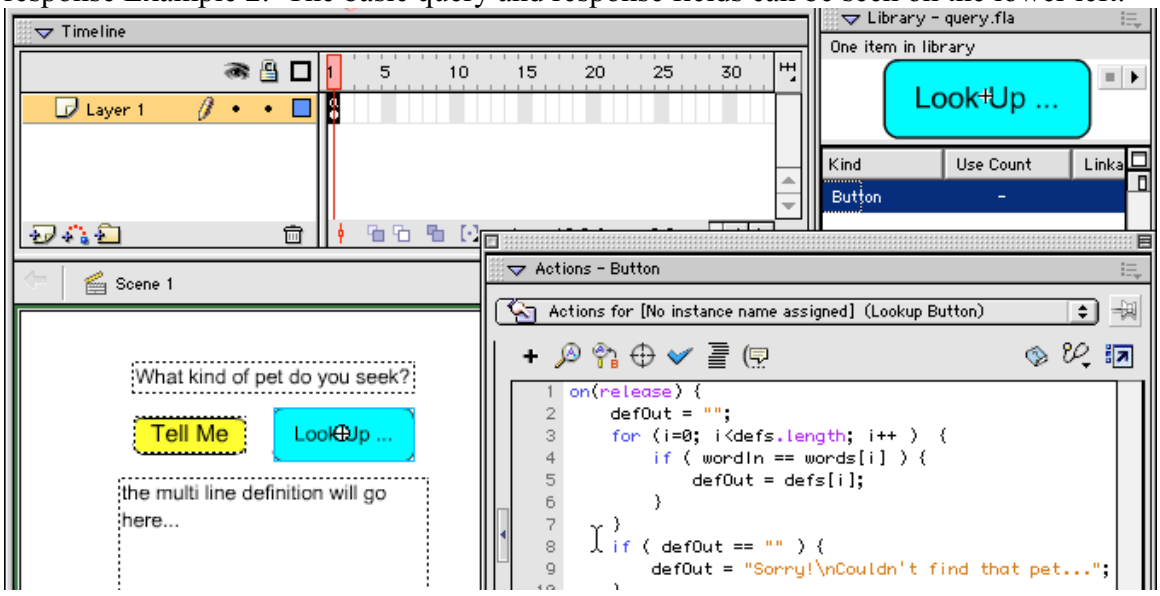
With the text tool, the student places two text strings on the stage, defines the properties of one as input and one as dynamic, and assigns variables to them. An enter query button is added to the library, and placed on the stage. This code is attached onto the button:

```
T  on(release) {
A     defOut = "";
A     i = 0;
A     while ( i < defs.length  &&  defOut == "" )  {
A           if ( wordIn == words[i] ) {
A                 defOut = defs[i];
            }
A           i++;
      }
A     if ( defOut == "" ) {
A           defOut = "Sorry!\nCouldn't find that pet...";
      }
}
```

In Frame #1, this initialization code is added:

```
A  words = new Array("dog", "cat", "gerbil");
A  defs = new Array(
A  "fluffy playmate, eats bones, likes to take owner for walk",
A  "pointy ears, chases birds, howls like baby when in heat",
A  "runs up sleeves when taken out of the cage"
);
```

Figure 2: Flash s ActionScript environment, showing the implementation of the query-response Example 2. The basic query and response fields can be seen on the lower left.



In the ActionScript implementation, outside of the initialization code, there are eight lines of algorithm code and one line of event code. There is a similar amount of algorithmic code in the Java version, but another 14 code statements are required. This additional scaffolding and windowing API code represents roughly an 50-50 split of scaffolding and API code to algorithm code in this exercise. This may be appropriate when the intent is to learn the windowing API, but not when the focus is on working with basic language constructs and algorithmic problem solving. Once again, the ActionScript is clear and concise; each line of code has a direct mapping back to specifics of the problem statement.

FRAMEWORKS FOR ACTIONSCRIPT-BASED CURRICULA

ActionScript is a complete language and supports general CS teaching needs. It has scoped variables with Boolean, numeric, and well-behaved string datatypes, and a full range of operators on those datatypes. It has `if-else` and `switch` conditionals, control constructs for `while` and `for` loops as well as enumerated `for` loops, and arrays. These all follow established C/Java syntax and semantics. The full range of user-interaction event handling is supported. A working set of OO constructs is in place, though advanced OO concepts such as abstract classes, interfaces, and multiple inheritance are not addressed.

We ve used ActionScript in a limited context of project-based work supporting science and mathematics classes, and the response was enthusiastic. It was easy to use and students produced engaging projects quickly. However, there are some rabbit-holes we discovered along the way that need to be avoided or controlled for the introductory use of ActionScript to succeed. Untyped variables proved not to be troublesome, however the unflagged use of undefined variables and propagation of the `undefined` value always required debugging effort to track down. In execution, an ActionScript fails silently without any indication of trouble — while quite irksome, this does improve ones debugging abilities and promotes incremental development. The splitting of code across the timeline, movie clips, and buttons needs to be controlled, with a solid rationale in place for a standard way to place the code appropriately. Colin Mook [5] notes,

> *Sometimes ActionScript is so flexible that it becomes slippery. There are so many ways to create a legal object-oriented application that finding a standard approach can become an arduous task. The following sample template for an object-oriented application attempts to provide some initial direction. You should adapt it to your needs...*

To use ActionScript successfully, its inherent flexibility must be constrained so the student or practitioner, at any level, can maintain conceptual control over their creation.

There are several ways ActionScript can be brought into our educational processes:

1. **Across the CS curriculum:** While possible in theory to use ActionScript as a sole language across all CS, this is not immediately likely. Nor is it desirable to force one tool onto all tasks, as our students should have a variety of tools and

conceptualizations at their disposal to solve problems in practice. There are certain programming aspects that require alternative languages to explain and explore.

2. **For the Introductory CS sequence:** This option will be discussed more thoroughly in the following sub-section. Table 1 maps out our existing set of topics for a first CS course and notes any necessary adaptations were ActionScript the *lingua franca* of the first classes. With a solid programming foundation, students entering further study would be ready for the richer and more structured development environments of Java or C++. We believe also that a gentle and engaging introduction to programming would spark more interest and help improve retention in our CS programs.

3. **In the distributive requirement class for those not majoring in CS:** Here is an especially appealing place for ActionScript, since these students should work in an environment which fosters engaging projects and avoids confusing scaffolding. There are no issues of language continuity in follow-on classes to consider. And some of these students will be fired up enough to consider a CS major, thus helping in CS recruitment. As a side benefit, students will gain experience with a tool popular in the web-development workplace.

4. **For project-based work in mathematics, sciences, and engineering:** Using a straightforward programming tool to visualize and explore lecture topics can be especially powerful in college and secondary school curricula as a way to integrate inquiry-based and project-based work into the classes. With a minimal introduction of programming constructs, we have had students in general science and mathematics create programs showing planets in motion and randomized simulations.

   As a simple example of this approach, adding motion in the y direction to Example 1, analogous to the x motion, is an easy extension with far reaching conceptual impact: the moving balls behave just as they would on a pool table, bouncing at appropriate angles off the cushion boundaries. After students simply observed this simulated behavior for a while in wonder, this then lead to a deeper discussion of decomposition of diagonal motion into vector components, and to a simple proof of the angle of incidence equaling the angle of reflection when bouncing off a surface.

Integration Into an Existing CS-1 Course: One Example

The following material is taken directly from our existing introductory CS course at Colorado State University, the Java-based CS153. This course is designed to teach students problem solving skills and programming. It assumes students have no previous programming experience.

CS153 has several high-level goals:

(1) To teach students how to decompose problems into smaller tasks and solve them
(2) To teach students the fundamentals of programming and specifically, the syntax of the Java language
(3) To teach students the concept of object-oriented programming development
(4) To teach students how to debug their own programs by printing information throughout the program
(5) To teach students good programming principles and standards used in practice
(6) To familiarize students with Eclipse, a graphical IDE for developing programs.

With ActionScript as the language in (2) and Flash MX as the IDE in (6), these goals are similarly appropriate. The following table summarizes the classroom topics by week, and the impact on the curriculum were ActionScript used in place of Java:

Week	Lecture Topic (current Java-centric topics)	Lecture Content if using ActionScript
1	Introduction, computers	same
2	Eclipse, Java syntax	Flash IDE, ActionScript syntax
3	Command-line arguments, formatting values	Not applicable. No need to process command-line arguments.
4	String manipulation	same
5	Writing classes	Similar, though less scaffolding necessary
6	Conditionals — if / switch	same
7	Comparing data types, Loops	Comparing data types is transparent and unnecessary; Loops would be the same.
8	Loops, Reading from files	I/O is from user screens, not files —straightforward and much easier to teach.
9	Pseudocode, Software Lifecycle	same
10	`static`, Interfaces, Overloading	`static` is similar to `global` in ActionScript; Overloading is analogous; Interfaces are Java specific.
11	Arrays	same
12	ArrayList, Inheritance	ArrayLists not applicable; cover Inheritance similarly.
13	Abstract classes, Polymorphism	No abstract classes with ActionScript.
14	Polymorphism, Sorting, Search	same
15	Search, Recursion, Exceptions	No exceptions in ActionScript, stress debugging instead. Search and recursion would be the same.
16	Final exam	same

Some points to note in this comparison:

- All the fundamental topics (except those specific to the Java-OO world) can be adequately covered with ActionScript, though some rearrangement of order may be appropriate. It might be argued that some of the advanced OO topics might be out of

place in this class, and indeed, the structure of the Java class is being changed along that line of thought.

- Removing the Java-specific scaffolding and Java-OO specifics such as interfaces frees up potentially three to four weeks out of a 16 week semester! It stands to reason that if scaffolding is no longer necessary and object creation simplified, the class time necessary for those topics is also greatly reduced. The freed class time can now be devoted to more thorough coverage of programming basics and the development of more interesting problems and case studies.

While this one example schedule shows a potential 20% efficiency improvement in classroom time, that does not totally come for free — these topics will need to be addressed if and when the student moves to another language. It should, however, be more efficient to cover them later on with the student at a higher level of programming sophistication.

ActionScript Curriculum Support

ActionScript is the programming part of Macromedia s Flash MX development package. The Flash software runs competently on both Windows and Mac platforms, under any of the recent flavors of operating systems. There are also open-source compilers available that generate SWF files (the ActionScript executable) without a need for the Macromedia package; these would be appropriate for more sophisticated students, but defeats the smooth development interface appropriate for introductory students.

Macromedia has a Macromedia Developer Teaching Solution package available for colleges and universities, where 35 Flash MX licenses are available for $700 along with one set of reference manuals, for a cost of $50/seat. This could be spread potentially over several classes if this were lab software. The package also includes their web page software Dreamweaver and a ColdFusion server, so costs could be shared with another group interested in web development work. For secondary school use, a K-12 site license for a school of 500 or more costs $3500, which amortized over two years is around $3/student.

There are several books on the market that teach ActionScript, though not specifically directed toward supporting an academic course on programming. These would be useful reference material as language resources. I have personally used the book by Mook [5] and would consider it appropriate for the classroom: it is one-half narrative (with good case-studies throughout) and one-half language reference. Additionally, while not supporting any specific pedagogy, a plethora of projects and exercises is available on the web.

SUMMARY

The ActionScript examples in this paper have demonstrated the potential for providing students with clear and concise programming exercises, uncluttered with the scaffolding and API coding necessary in Java or C++. The ActionScript environment has several aspects that make it an appropriate choice for introducing students to programming:

- students can focus on learning basic language constructs and algorithms, not befuddled by scaffolding and API code;
- the implicit timeline is conducive to developing visually engaging animation and interactive exercises;
- the language constructs follow the currently leading languages Java and C++;
- the graphical nature of object creation and instantiation can be used to provide a gentle, concrete approach to an object-first pedagogy;
- a gentle programming approach with a kinder learning curve may improve retention and recruitment in CS programs;
- students are learning a language and environment that is immediately marketable in web design and development firms.

REFERENCES

[1] Cooper, S., Dann, S., & Pausch, R., Alice: a 3-D tool for introductory programming concepts , *CCSC '00: Proceedings of the Fifth Annual CCSC Northeastern Conference on The Journal of Computing in Small Colleges, 2000*, pp.107-116.

[2] Dale, N., Language wars , Panel Session position statement, 2003, http://www.cs.utexas.edu/users/ndale/, retrieved 10 April 2005.

[3] Dale, N., Content and emphasis in CS1 , to be published in SIGCSE Bulletin, 2005.

[4] *Final Report of the Joint ACM/IEEE-CS Task Force on Computing Curricula 2001 for Computer Science*, 2001, http://www.computer.org/education/cc2001/final/index.htm, retrieved 10 April 2005.

[5] Mook, C., *ActionScript for Flash MX; The Definitive Guide*, San Francisco, CA: O Reilly & Associates, 2003.

[6] Reed, D., Rethinking CS0 with Javascript . in *SIGCSE'00 ACM Digital Library* , 2000.