ABSTRACT

We present the conceptual design and first-phase implementation of an approach to computer programming instruction that has been designed to exploit the attractive force of video games. We partition the skills that a professional programmer must master into a sequence of eight levels, where each level introduces new programming tasks that are to be implemented on a progressively more detailed machine architecture. Our approach is intended to foster a conceptual understanding of the internal representation of data, and the underlying operations being performed on the data by an instruction sequence. To motivate the student programmer, our approach is designed to be integrated into a video game where, to progress through the levels of the game, the player must also progress through the levels of programming tasks. After completing all levels the player of such a game would have a thorough mastery of fundamental programming skills, including array indexing, recursion and pointer variables, coupled with a solid understanding of the underlying computational model and the von Neuman architecture.

1. INTRODUCTION

We are developing an approach to teaching computer programming that leverages the attractive force of video games to motivate and facilitate acquisition of the skills and understanding necessary for computer programming.

Computer programming is an exacting task that requires the programmer to master the rigorous syntax and semantics of a computer language, and to understand the underlying architecture on which the program will execute. In addition, the fledgling programmer must begin to master the paradigms of programming, including the algorithms and structured thinking required to formulate a programmatic solution to a problem [1].

To address the difficulty in learning to program a computer, researchers and educators have developed a number of specialized programming languages and environments. Most of these approaches, including Logo, Karel the Robot, Alice and ToonTalk [2, 3, 4, 5, 6, 7, 8], create a programming environment that removes the syntactic rigor and much of the semantic difficulty of learning to program; they build an abstraction of a language and provide drag and drop operations, typically expressed as objects or icons, that make computer programming accessible to elementary students. These approaches substitute a virtual machine with higher level semantics for the von Neuman architecture of the real machine. The strength of these approaches is that they make computer programming accessible to nonprofessionals, including elementary students. Their weakness is that the machine abstraction hides the true architectural model, which a professional programmer must comprehend and master.

We believe that a very different approach is needed for training students who will become professional programmers. We are developing a conceptual design that fits programming instruction into a gaming environment where student programmers are guided through a progressively more complete instruction set and machine architecture until the full complexity of the machine’s programming interface is exposed. During this process, the student must develop instruction sequences that solve programming challenges that become increasingly more complex. Thus, there are two components in our approach: an architecture simulator and a video game. The focus of this paper is the architecture simulator, including a full exposition of various levels of progression in programming skill incorporated into the simulator. We provide some guidelines for the design of a video game that will seamlessly interface with the architecture simulator; however, a full description of an actual game remains future work.

The architecture simulator features real-time animations to illustrate the actions that are generated by the instruction sequence developed by the user. As the user program executes, the animation illustrates both control flow and the movement of data between input, memory, registers, and output. A key feature of this simulator is that is is designed to gradually “unveil” the underlying complexity of the full machine architecture as the student’s grasp of concepts develops. This is intended to foster a staged conceptual understanding of the internal representation of data, and the underlying operations being performed on the data by an instruction sequence.

We have partitioned the programming skills that a fledgling programmer must develop into a sequence of eight levels, where each level introduces new programming tasks that facilitate acquisition of new skills, as well as fostering an understanding of the underlying computer architecture. Our
current implementation of the architecture simulator is intended as a proof of concept and includes only the first of these eight levels. In this paper we describe the conceptual design of all eight levels. The eight levels progress in complexity as more of the underlying architecture is exposed; this is analogous to the levels of a video game that increase in complexity as the player progresses through the game.

To motivate the development of programming skills, we will incorporate the programming tasks into the context of a video game. The game can assume any of the popular genres, but we envision a role-playing game where the player assumes the role of a character that lives inside of a computer and must struggle against computer viruses, robots and other enemies. To overcome the enemies, the player must solve puzzles that are generated by the enemies, or that populate the game as an artifact of enemy activity. For example, completion of a programming task involving one register might earn an additional register for the player to use in solving subsequent programming puzzles.

This progression through both the video game and the simulator provides a mechanism of assurance that the player has mastered programming concepts at one level before advancing to the next level. In this way, the cognitive demand on the player will be automatically adjusted to his or her learning rate. More importantly, as players progress further in the levels of programming skills, they also develop a deeper understanding of the computational model inherent in the von Neuman architecture.

This approach accommodates various learning styles and capacities and does not depend on a specific game to engender student interest. Rather, our modular approach permits the development of a variety of game shells that embed our framework. There is no approach described in the literature that exploits the attractiveness of video games in this plug-and-play fashion. Finally, our approach is designed to nest identical curricular content frameworks within different game shells. This approach facilitates tailoring of game play to suit a target audience, while maintaining the same learning content across games.

In the next section we describe previous approaches to programming instruction and compare them to our approach. In Section 3, we provide detailed description of our approach, including an overview of the architecture simulator and a summary of the eight levels of architecture, which progressively removes the veil over the underlying computational model. In Section 4, we describe our architecture simulator that tracks the essential actions of the computer as each instruction executes. Finally, in Section 5, we draw conclusions and describe our ongoing work.

2. PREVIOUS APPROACHES TO INSTRUCTION ON COMPUTER PROGRAMMING

One of the oldest and most respected approaches to the instruction of computer programming is Logo, a functional programming language developed in 1967 by Wally Feurzeig and Seymour Papert at Bolt, Beranek and Newman (BBN) [2, 3]. One of the most important features of Logo was the turtle graphics that enabled children and novice programmers to control a virtual turtle moving around an environment using simple function calls that provide immediate visual feedback about the results of the call sequences. Logo supports data structures such as lists, and control structures for decisions and loops. Thus, while Logo’s simplicity and graphics appeal to young users, its sophisticated programming constructs provide power for more advanced learning.

A key feature of Logo is the abstraction provided by the simplified function calls, which remove much of the syntactic complexity of the programming language and the architecture from user consideration. This feature of Logo is based on the assumption that actual language syntax and semantics place a heavy cognitive burden on the user, who is either not ready or incapable of understanding what is happening “under the hood” of the computer.

Karel the Robot, developed in 1981, is similar to Logo [4]. The Karel language consists of only 5 basic instructions and these instructions are used to move the robot, Karel, through a rectangular grid of streets and avenues. The programmer uses these instructions, together with control statements if, while and iterate, to control Karel.

The language Alice, developed at Carnegie Mellon University [5], is similar to Logo and Karel in that Alice abstracts away the normal language syntax to enable the user to build 3D animations. However, Alice goes further by combining an integrated development environment (IDE) with drag and drop icons to bring visual and 3D programming to groups that are not normally exposed to programming. A recent version of Alice, Story Telling Alice, was developed to permit the user to create animations containing interactions among the characters [9].

ToonTalk is an interactive world that resembles a modern city in which a carefully designed sequence of puzzles introduces the user to programming concepts and techniques [7, 8]. It consists of typical city objects such as houses, streets, cars, trucks, birds and boxes. Houses contain robots, which perform most of the actions for solving problems in the ToonTalk world. The robots can be trained to perform small tasks by entering commands into the “thought bubbles” above each robot’s head. The canonical programming concept exemplified in ToonTalk is swapping the values of two locations: the robot is instructed to grasp the contents of one location, set it down, grasp the contents of the other location and place it in the first location, and then to grasp the original item and move it to the second location.

3. LIFTING THE VEIL

In this section, we describe our approach toward instruction in computer programming and provide details about the sequence of eight levels of programming tasks that progressively expose information about what actually happens when computer programs execute. Our goal is to incorporate these programming tasks into a video game. Solutions to the puzzles are intended to be used to permit the player to make faster progress in the game, or provide incentives to the player in the form of powerups, more powerful weaponry or additional information.

The intent of our design is that as the player progresses through the video game levels, he must also progress through the levels of programming puzzles. As the levels progress and the puzzles increase in complexity, additional language constructs and architectural features are added until, finally, the entire organization of a computer program is presented as it is laid out by a typical compiler. As registers, memory and instructions are added to the architectural model and the programming assignments increase in complexity, the student programmer will make progress in mastering ar-
rays, recursion, and pointers in a sandbox environment that provides guidance in avoidance of many common programming errors.

### 3.1 Level I: Programming Fundamentals

Figure 1 illustrates the initial architectural model, presented to the student at Level I. The student uses only four registers and five instructions; however, there are a myriad of important program specifications that the nascent programmer can master with this reduced instruction set, including swapping two integers, reversing two integers and finding the largest of an arbitrarily sized set of integers. At this level, the student is armed with five instructions, providing a large enough arsenal to master the fundamental concepts involved in I/O, loops and decisions. In addition, at this level, our simulator uses animations to illustrate the manner in which data is moved from registers as the instructions execute. The animations serve to clarify the real activities that occur when instructions execute.

![Figure 1: Level I consists of only four registers.](image)

### 3.2 Level II: Array Indexing

At Level II, we augment the four registers from Level I with indexed memory, illustrated as E in Figure 2. Indexed memory serves as an analogy to an array, which can present difficulties for the fledgling programmer. The presence of the array structure enables the introduction of the notion of run-time determination of the stored data set size. This allows us to introduce new program specifications: for example, reading a sequence of integers until an end sentinel is encountered, printing the sequence in reverse order, and sorting a sequence of integers.

![Figure 2: Level II adds an array.](image)

### 3.3 Level III: Type Systems

At Levels I and II, the student programmer developed code that used only integer variables without the encumbrance and protection of type declarations. The design of Level III introduces the notions of variable naming and types attached to variables, permitting the declaration of variables of type int, float, and char. Figure 3 illustrates, on the left, a sample program that might be developed at Level III, and the corresponding architectural view on the right. In the sample program, the user has declared two integer variables, i and j, an array f of floats, and a character variable c. On the right side of the figure, the allocated memory cells are illustrated with labels to the left of each cell, indicating the declared type of the variable. The value assigned to a variable is shown inside its cell. Consistent with the homogeneous nature of arrays, the simulator assigns the same type value to all array locations.

The introduction of type information also introduces a great number of new problem specifications that can be posed to the student programmer. For example, problems about integer division, modular arithmetic, and character strings can now be introduced. Although the notion of integer truncation can be difficult for student programmers, the architecture simulator provides visual feedback when solving such problems as computing a remainder.

### 3.4 Level IV: Contiguous Memory

In the design of Level IV, we introduce the important concept of memory, that each variable declared in the user program is assigned both a type and an address, and the address of a variable corresponds to a location in memory. We also introduce the fact that memory is a contiguous sequence of locations and that each location has an address. In this gentle unveiling of the von Neuman architecture, we assume that the contiguous sequence of memory locations begins at location 0, and that all variable types fit into one cell of memory.

Figure 4 illustrates two possible views of the mapping between variable names and memory locations that can be shown to the user. For Figure 4a, a sample program is shown on the left, variables i, j, f and c are shown in the middle, and the corresponding contiguous memory locations are shown on the right. The important concept that we introduce here is that the values assigned to these variables are not stored in the symbol, but rather the symbol indicates where the actual values are stored. For example, the value 1 in the box for variable j corresponds to memory location 1, shown on the right side of the figure, where j’s actual value of 3 is stored.

Figure 4b illustrates an alternate display of the same information where, rather than storing memory addresses in the variable symbols, the simulator draws arrows to the memory locations to visually indicate the mapping between variables and memory locations. This illustrates a key feature of our simulator design, which is to provide, whenever possible, alternate views of the underlying architecture, including permitting students to revert to previous architectural views. We do this to permit students to return to their comfort.
zone if, at any time, they become confused by a new representation.

### 3.5 Level V: Internal Representation of Data

The design of Level V shows the student that the actual memory layout is in the form of individually addressable bytes, and that different variable types use different numbers of bytes. We also introduce the notions of binary numbers and hexadecimal representation. At this level, the student programmer learns the stark difference between the internal representations of the three data types that we introduced at Level III. Students can begin to see why a variable of type `float` can be much larger but not as precise as an integer, since the visual representation illustrates this difference. The student also begins to see that the internal representation of character data takes only a single byte. Of course, the introduction of the internal representation of character data necessitates that we also introduce the concept of byte manipulation, since character data does not require a full computer word.

Figure 5 illustrates the extension of the architecture to include the internal representation of data and therefore has the same user program that used in our previous discussion. However, the figure now shows that the memory addresses for the user declared variables are expressed in hexadecimal, and the right side of Figure 5 shows that the internal representation of data for each computer word is now partitioned into four bytes, and the memory addresses of each computer word are now listed in hexadecimal. For example, the symbol information for variable `j` now shows that the value for `j` is located at memory address 4. Hopefully, the student can now see that the value for `j[2]` is stored at memory location `C` and that the internal representation of the data is `00 0000 03 40 BE17 77 79 3D C275 8F C0 C440 9C 43 0052 00 00 0000 03`.

Figure 5: Level V. Byte structure and hexadecimal exposed

### 3.6 Level VI: Functions & Activation Records

At level VI, we introduce the important concept of programming using functions. We represent a function invocation as a tabbed rectangle, where each tabbed rectangle represents an activation record. An activation record appears for each function invocation and activation records are stored on a run-time stack called the control stack [10][p. 433]. The contents of activation records vary with the language being implemented; however, each record must store enough information to fully represent each procedure invocation. In our approach, we gradually expose more information about function activations.

Figure 6 illustrates our initial representation of functions and their corresponding invocations and activation records.
and an internal representation of the corresponding data values and memory locations on the right side of the figure. The user program on the left side of Figure 6a contains three functions: bar, foo, and main. Execution of the user program begins in main and the current state of the simulator indicates that the user has made a call to foo; thus, there are two records on the control stack: a record for main at the bottom of the stack and a record for foo at the top of the stack. Also, the simulator tracks the current value of the program counter and indicates its current value with an arrow pointing to the signature, the first line, of function foo.

The activation record for foo, shown in the upper left corner of Figure 6a, conveys important information about function invocation: all values for local variables and value parameters must be stored locally, in the activation record for foo. Our representation of each record as a tabbed rectangle illustrates the essential nature of a stack: the only record that is available is the record on top of the stack.

Our memory layout at Level VI is illustrated on the right side of Figure 6a, where we use the notation main::i to designate the location corresponding to the variable i in main. Our use of the C++ scope operator is intentional and will facilitate the transition to object-oriented programming. An important consideration here is that the value, currently stored at location 0, main::i, is undefined since the activation record for function foo is still active and a value has not been returned yet for variable main::i.

Figure 6b further illustrates the actions of the control stack, with an activation record for function bar at the top of the stack. The arrow in the figure indicates that execution has reached the end of bar and that a value has already been placed in memory location 1C, on the right of Figure 6b. Our architectural simulator is illustrating to the student programmer the technique used to return values from function invocations. As execution continues, the activation record for bar will be popped from the stack, a value for k will be computed for function foo, and eventually, a value will be returned for main::i, and that value will be entered into memory location 0. This treatment of the control stack mirrors the manner in which function activations are handled in the actual compiler implementation for functions.

### Table 1: Level VI introduces procedures and scope.

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>int m</td>
<td>n = 2 * m; print m; return n;</td>
</tr>
<tr>
<td>foo</td>
<td>int m, int n</td>
<td>n = 2 * m + g; k = bar(m) + n + f; return k;</td>
</tr>
<tr>
<td>main</td>
<td></td>
<td>i = foo(j, k);</td>
</tr>
</tbody>
</table>

### Figure 7: Level VII introduces global and static variables.

#### Global Variables
- g
- C
- f
- m

#### Static Variables
- f
- m
- g

#### Figure 6b: At the end of bar

#### Figure 6a: Upon entrance to foo
The important consideration at Level VII is that the static variable is stored with global variable $g$ because subsequent invocations of $foo$ will use this same storage and $f$ will retain its values across invocations to $foo$. However, storage for parameters and local variables of $foo$ are stored after the variables for function $main$. Thus, our simulator illustrates the dynamic nature of function invocation and the corresponding local and value parameters, whereas storage for static variables is similar to storage for global variables, but the scope of static variables in functions is local to the function.

3.8 Level VIII: Pointer Variables

At Level VIII, our design is to present the full memory model as it might be laid out by a typical compiler, including the introduction of dynamically allocated memory, pointer variables and reference parameters. Figure 8a shows a typical user program for this level and Figure 8b shows the corresponding memory model. The user program contains five functions: $swap$, $readnums$, $writenums$, $sortnums$ and $main$. Function $swap$ illustrates reference parameters $n1$ and $n2$ and the arrow at $swap$ shows that execution is currently in this function; the user code for $swap$ illustrates the traditional 3-step swap. We use the C++ notation, &, to indicate reference parameter transmission. Function $readnums$ has two parameters and the first parameter is a dynamically allocated array, $table$, which provides storage for the numbers to be sorted. $table$ is also a parameter to $writenums$ and $sortnums$.

Figure 8b illustrates the memory model for the user program in Figure 8a, where a control stack is shown in the upper left, with function $swap$ at the top of the stack. A record for global variables $global1$ and $global2$ is shown beneath the control stack, and the current state of memory is shown on the right of the figure. Our memory model illustrates storage for both the heap and the control stack and these data structures grow toward each other. Storage for the two global variables is shown in the lower right of the figure. The heap, shown at the top of the memory model, shows that storage has been allocated for eight integers. The large purple window to the right of the user code is the output window; this transmission is informed by execution of an instruction and the currently executing instruction is indicated by the arrow on the left side of the user code window.

After the player completes a proposed solution to a programming puzzle, the simulated execution of the code can be used either to verify that the solution is correct, or to facilitate the debugging process. Typically, the simulated execution of an incorrect solution will provide feedback about where the fault lies. After successfully solving a puzzle, or after failing to solve a puzzle, the player may request that a sample solution be presented in the user code window. The student solution illustrated in Figure 9a uses three if statements and four comparisons to find the largest of three numbers. However, this solution is error prone and does not easily scale to more than three numbers. Figure 9b illustrates a better solution to the problem of finding the largest of three numbers. This solution scales to more than three numbers and is less error prone than the student solution.

5. CONCLUDING REMARKS

We have described the design and preliminary implementation of our approach to computer programming instruction using a gaming environment. Unlike previous approaches that build an abstraction of the architecture to hide details and make programming accessible to non-professionals, our approach progressively exposes the details of the underlying computational model. Our target audience is the student who is interested in becoming a professional programmer and therefore needs to master the underlying computational model. Through a succession of eight levels, we introduce features of the architecture and corresponding memory model until the full model is exposed, including arrays, recursion and pointer variables. We have described a prototype implementation of Level I of our architecture simulator and we have illustrated its use in solving programming challenges.

At the end of a game, our hypothesis is that the student programmer will have acquired full comprehension of the computational model that forms the cornerstone of professional computer programming. We are currently working on the design and implementation of a video game to be used with the architecture simulator and our future work
6. REFERENCES


(a) user program

int global1, global2;

int swap(int n1, int n2){
    int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
} void readnums(int *table, int n){
    int i;
    i = 0;
    while(i < n){
        input(table[i]);
        i = i + 1;
    }
}

int main(){
    int n;
    int *table;
    input(n);
    table = new int[n];
    readnums(table, n);
    sortnums(table, n);
    writenums(table, n);
    return 0;
}

void sortnums(int *table, int n){
    int low;
    int i;
    low = 0;
    while(low < n - 1){
        i = low + 1;
        while(i < n){
            if(table[i] < table[low]){
                swap(table[i], table[low]);
                i = i + 1;
            }
            low = low + 1;
        }
    }
    int *table;
    input(n);
    i = 0;
    while(i < n){
        output(table[i]);
        i = i + 1;
    }
}

void writenums(int *table, int n){
    int i;
    i = 0;
    while(i < n){
        output(table[i]);
        i = i + 1;
    }
}

(b) memory model

Figure 8: Level VIII introduces dynamic memory allocation.
(a) Finding the largest: Brute Force

(b) Finding the largest: Advanced Approach

Figure 9: The Architecture Simulator for Level I