More than the Usual Suspects: The Physical Self and Other Resources for Learning to Program Using a 3D Avatar Environment

Kate Starbird
ATLAS Institute
University of Colorado at Boulder
catharine.starbird@colorado.edu

Leysia Palen
Department of Computer Science
University of Colorado at Boulder
palen@cs.colorado.edu

ABSTRACT
This paper presents results from a video-based analysis of non-programmers' use of a new platform for end-user programming, the 3D Avatar Programming System (3DAPS). We use micro-ethnographic analytic methods to understand how learning about programming occurs. We discuss how the management of internal and external cognitive representations of 3D movement information leverages existing, embodied knowledge to unravel less familiar knowledge—that of programmatic instruction. In other words, the 3D movement serves as the language of translation between the representations to support learning. We also examine how shared code is used as an educational resource in a learning environment without a teacher.

Categories and Subject Descriptors
K.3.1 Computer Uses in Education - Collaborative learning; K.3.2 Computer and Information Science Education; H.5.2 Information Interfaces and Presentation—User Interfaces (D.2.2, H.1.2, I.3.6); D.2.6 Programming Environments - Graphical environments.

General Terms
Design, Human Factors, Languages

Keywords
Avatars, broadening participation, computer science education, constructionism, distributed cognition, end-user programming, interaction analysis, online-identity, social networks.

1. INTRODUCTION
Computer science educators and education researchers have developed a variety of end-user programming systems aimed at teaching computer-programming skills to novices [11]. In this paper we present results from a video-based analysis of the use of a new platform for end-user programming, the 3D Avatar Programming System (3DAPS), designed to include features that would further appeal to would-be programmers—that of programming an avatar as a personal representation of self that could then appear in different social networking forums.

iConference 2011, Feb 8–11, 2011, Seattle, WA, USA.

Here, we examine how a target audience uses such a system, not as a simple matter of identifying usability problems, but rather to formulate understandings about programming learning, with a longer term hope of broadening participation in the computing field. After setting up a research design that would allow for extended engagement with the programming environment, a detailed video analysis combined with log data analysis revealed learning patterns that built on the ability to share code with known and unknown others. It also unexpectedly revealed learning behaviors that relied on *embodied cognition* to translate the seemingly esoteric activities of programming into more personally meaningful action.

1.1 Prior Work & Design Rationale
This research works from the premise of the popular end-user programming platform, Alice—that 3D content generation can be a motivational draw for novices to computer programming [6, 12]. The 3DAPS platform allows users, ideally novice programmers, to control the actions of a three-dimensional Avatar.

In addition, long-standing research in end-user programming and education holds that the sharing of personal creations supports powerful *constructionist* learning experiences [5, 14, 15]. Another existing end-user programming system, Scratch, exploits this phenomenon in a socially-networked, Web 2.0 environment, leveraging sharing for peer support and as motivation to participate [16] The 3DAPS platform incorporates a dynamic library that allows users to share, borrow, and modify action code.

1.2 Research Objectives
This research explores how 3D content and sharing can be used as more than mere motivation for participation within end-user programming platforms – but also as resources through which learning occurs. Brandt et al. show how current programming practice incorporates code-sharing [4], and investigate methods for supporting this behavior [3]. In this paper, we show how novice programmers, within an environment that has no "teacher," use borrowed code in different ways to learn computer-programming skills.

In addition, this investigation unexpectedly uncovered how novice programmers mapped 3D graphical representation to the actions of their own bodies to serve as a resource for programming. Alac reports on a similar phenomenon, showing how scientists learning how to program a robot used their own bodies to understand, visualize and control the movements [1]. In her work, the connection between self and robotic representation of self, as well as the scientists' exploration of motion within these two different

forms (what Alac called *body-in-interaction)*, was integral to the learning process. The results of this research reveal forms of embodied learning, and show how the connection of physical self to a digital representation of self (a 3D Avatar), can drive learning within an end-user programming environment.

2. PROGRAMMING ENVIRONMENT

The 3DAPS working prototype contains several key design components to allow for focused evaluation of both current functionality and feasibility as an educational tool. The focus of this first major phase of research was to isolate and evaluate conceptual building blocks of the full-fidelity system. Though future design directions will investigate drag-and-drop and graphical programming, the current system uses a text-based approximation of JavaScript, intended to make programming more user-friendly, but flexible enough to serve as an easily modifiable research tool.

2.1 3DAPS System Design

3DAPS consists of four components: the Programming Environment, the 3D Graphical Display Window, the Construct Window and the Library Window (Fig. 1). The Programming Environment is a textual window that contains code that runs a motion script created by a user. The Graphical Display Window contains a 3D Avatar, currently composed of simple shapes. Scripts created in the programming window animate the Avatar in real time. The Construct Window provides textual examples of different programming constructs (loops, conditional statements, etc.) that users can use for reference, or copy/paste into their programming window. The Library Window contains the textual code of previously created functions that have been shared with the library. Users can download functions from the library into their Programming Window using copy/paste and see the motions they create using a simple function call. Users can manipulate downloaded code in any way they wish, changing existing functions or copying/pasting any piece of code into their own functions. They can then share their motion and its code with the library for other users to see and use. The <Share> button was simulated wizard-of-oz style during the user studies.

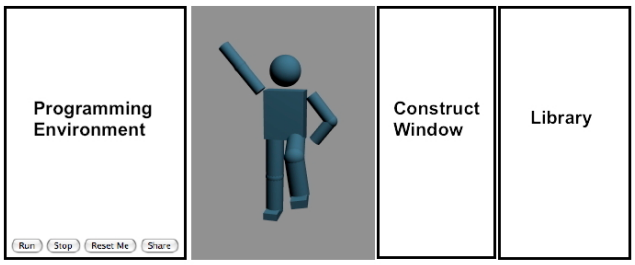


Figure 1. 3D Avatar Programming System layout. Buttons at bottom left say *Run*, *Stop*, *Reset Me*, *Share*.

2.2 System Specifications

The Programming Window and the Graphical Display Window exist as a single webpage run through a web browser. The system is implemented in JavaScript using Google's O3D plug-in to render the 3D graphics in real time. O3D provides graphical support and compatibility with several 3D modeling systems that will enable the future system to animate high-quality 3D Avatar models. For these studies, the Construct and Library Windows were simple text windows. In future implementations, constructs will be drag-and-droppable and the library will feature one-click testing and downloading.

2.3 Programming Language

The 3D Avatar programming language resembles a limited form of JavaScript with similar syntax. At its base are three types of Basic Movement Functions that users can call:

```
Move(x, y, z, time);
Rotate(x, x, y, time);
RotateLeftHip(x, y, z, time);
```

The Move command moves the Avatar *x* units along the X-axis (left/right), *y* units along the Y-axis (up/down), and *z* units along the Z-axis (forward/backward) in *time* seconds. The Rotate command rotates the Avatar *x* degrees around the X-axis, *y* degrees around the Y-axis, and *z* degrees around the Z-axis in *time* seconds. The RotateJoint commands take the same arguments as the Rotate command. As with a human body, a rotation of a joint will rotate all of the bones and joints below that joint, as well as their axes of rotation.

There are two movement constructs built into the Avatar Programming Language. The DoInOrder Movement Construct specifies a list of motions that should be executed in order, one at a time. The DoTogether Movement Construct specifies a list of motions that should be executed all at the same time. The latter construct, for example, could be used to make an Avatar move forward and swing both arms and legs at the same time in a walking motion. Movement Constructs can be nested inside each other and can contain other constructs and any other combination of code. We borrowed the DoInOrder and DoTogether terms from the functionally similar Animation Objects in the Alice programming environment [6].

In addition to the Basic Movement Functions and Movement Constructs, 3DAPS supports a limited range of programming constructs: integer and float variables, arithmetic expressions, conditional expressions, if-then-else statements, for-loops, while-loops, functions, arguments, return variables, and a random number generator.

2.4 Note on 3D Programming in 3DAPS

Whereas original Alice designers attempted to make 3D more accessible to novices by removing the concepts of axes, degrees, rotations and translations from the programming language [6], 3DAPS exposes these complex interactions in 3D graphics. We hypothesize that 3D problem-solving could support programmatic learning specifically and computational thinking generally, a topic we return to later in the paper.

3. RESEARCH DESIGN

The user studies are designed to evaluate interaction with the early prototype and to assess qualities of learning using a 3D avatar-based programming environment.

3.1 Participants

We initially recruited three pre-college or early college non-programmers who were each asked to find a friend to participate in the study. The purpose of the pair configuration was to encourage verbal communication of the participants' programming strategies and other reflections. The three pairs are

denoted as Groups A, B and C. One participant (Greg[1] from Group A; see Table 1 for full listing of participants) was unable to continue after the first session, and was replaced by Amy, who had participated in the pilot study and therefore had the same 3DAPS experience as other participants entering Session 2. Amy's programming skills were comparable to other participants.

	SESSION 1	SESSIONS 2 & 3
Group A	Sandra, Greg	*Sandra, Amy*
Group B	Jorge, Mali	Jorge, Mali
Group C	Paul, Lisa	Paul, Lisa

Table 1. **Participants by Session.** (Note Group A changes.)

3.2 Research Design

Each pair completed three one-hour sessions over alternating days in a single week.

3.2.1 Library States

The library was initially seeded with five complex movement functions, demonstrating, minimally, all possible constructs in the 3DAPS language. To simulate a dynamic library, new functions were added over the course of the study. Additions by researchers were made between session days, while participant-created functions were added immediately following each pair-session for use by others in subsequent sessions. Participants could interact with these shared code contributions and their resulting animations, simulating a networked programming community. This also permitted us to observe interactions with a dynamic system over time.

3.2.2 Preemptive Syntactical Help

Due to the rigidity of the language syntax and because future designs will not rely on syntax at all, researchers worked throughout the study to provide preemptive syntactic help to study participants.

3.2.3 Session Design

Session 1 began with a short demo and a set of eight tasks designed to familiarize participants with the system layout, Basic Movement Function calls, and interacting with the Library. The remainder of the session was devoted to free time for participants to "play with the system."

Session 2 also began with a demo (on a separate machine) of motions created by researchers and added to the library after Session 1. Next we instructed participants to spend the rest of the session creating a motion or motions to share with the library. Also during this session researchers prompted participants to think about making part of their routine "repeat, 50 or 100 times." This large number was intended to push participants towards examining and attempting to use the loop construct, so we could analyze how they acquired expertise to use new constructs.

Session 3 for each pair was divided into three parts: a short briefing, free programming time, and a debriefing.

3.3 Data Collection

Two digital video cameras recorded participants' interactions with the machines and each other. The Camera Side video was at a high enough resolution and positioned to be able to determine participants' eye gaze toward each of the two monitors, the keyboard, paper notes, and each other. Additionally, we used screen capture software on both machines to generate video of all their computer interaction activity, including screen state, code changes, mouse locations, mouse clicks, and Avatar motions. Audio was captured in four feeds from both cameras and both computers (See Fig. 2) yielding excellent audio quality even when participants spoke quietly as they were thinking.

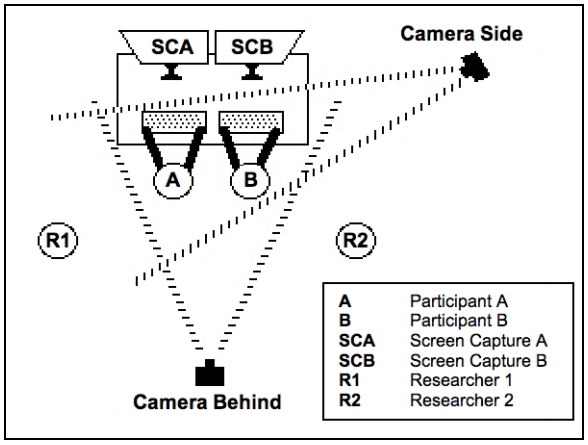


Figure 2. Physical layout of the user testing area

3.4 Methods of Analysis

For analysis of participant activity across multiple audio and video feeds, we used the Digital Replay System (DRS), software that supports qualitative analysis and simultaneous reply of multiple sources and data formats [9]. DRS allows synchronization of video onto multiple, overlapping video tracks and qualitative analysis by laying down associated *qualitative coding* tracks[2].

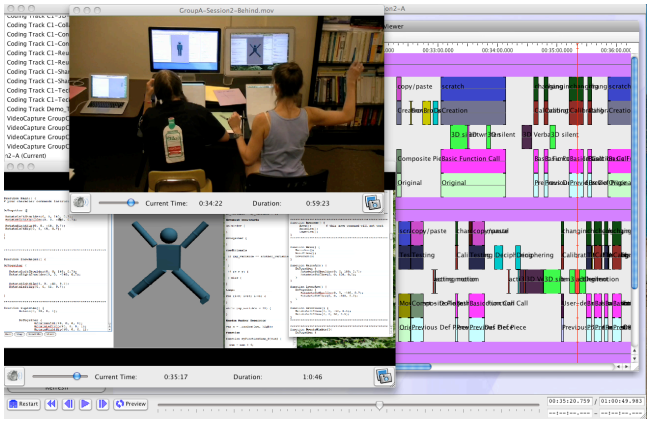


Figure 3. Screenshot of the DRS analytical environment.

We derived a qualitative analytical coding scheme based on the data. In a second viewing pass, every programming action for each participant was analytically coded for behavior type, re-use location, construct used (Basic Function Call, loop, etc.), and code-generation technique (typing from scratch, copying-and-

[1] All participant names have been anonymized.

[2] It is important to note the difference between *programming code* created by programming activity and *qualitative codes* derived from and used during qualitative analysis.

pasting, changing an existing line). Behavior types initially included testing, creating, modifying to test, modifying to create, calibrating, borrowing, browsing, and downloading. The de-coding category was added after further analysis. Re-use location described the type of code that a participant was working on: an original creation, a borrowed piece, or a borrowed function. Additionally, participant behavior was coded for demonstrations of 3D-related verbalization, physical enactment or gesturing to their programs in reference to some aspect of 3D problem-solving. Pair interactions were coded for instances of collaborative behavior. For selected "moments" that illustrated noteworthy behaviors, we created verbal transcripts with supporting physical behavior transcription, and associated these with representations of the step-by-step programmatic activities of the users.

3.5 Data Representations in this Paper

The presentation of data excerpts in this paper is modeled after Flor & Hutchins' reporting of code development during pair a programming exercise [8] and Crabtree & Rodden's reporting of diverse interaction and data types in hybrid ecologies [7]. We combine the three elements of verbal, physical and programmatic activity in succinct representations to best accommodate the constraints of manuscripts. Researcher observations and descriptions are presented in Helvetica 8. Verbalizations are marked within those sections in *Italics.* Coding excerpts are labeled and highlighted in boxes.

4. FINDINGS

Analysis of participants' interactions within the 3DAPS system provides insight into the possibilities of leveraging code-sharing and embodied interaction as learning resources.

4.1 Patterns of Programmer-Code Interaction

For each participant, we classified every discrete interaction with the system as to the code-generation technique and type of behavior exhibited (testing, creating, modifying to test, modifying to create, calibrating, borrowing, browsing, and downloading). From these codes, a few patterns emerged – series of three or four actions that often occurred in the same sequence. These patterns (*creating, exploring, borrowing, tinkering,* and *de-coding*) access different resources of the system and generate different types of learning experiences.

4.1.1 Creating

The *creating* pattern consisted of writing or copying/pasting a new movement command, testing that command, evaluating the results against a desired motion, and then either calibrating the existing movement (by adjusting rotation numbers) or beginning again by adding another new command. When *creating*, participants generated their own motions from scratch and did not rely much, if at all, on the library for examples. They used the results from their own motion commands to interpret how the system worked and what each command did. Participants who stayed primarily within this pattern (Amy and Paul) were slower to incorporate more complex constructs (loops, variables, etc.) into their programming practice than users who demonstrated other patterns.

4.1.2 Exploring & Borrowing

When *exploring*, participants downloaded functions from the library, and executed them within their graphical environment to see what they did. Some users then incorporated whole sections of borrowed code into their creations (Sandra, Paul, and Lisa). Though these activities allowed users to see the potential of the system, they did not appear to lead to increased understanding of programming.

4.1.3 Testing and Tinkering

The testing or "tinkering" pattern included borrowing existing code, then changing numbers or the order of components to "see what happens." Jorge used this behavior almost exclusively, and from his verbalizations, it was clear that he was attempting to derive understanding of how the system worked from the motions that resulted from his changes. On multiple occasions, Jorge described his behavior of this kind as "tinkering."

Illustration 1: Rationale for Tinkering

This illustration provides insight into how and why some participants used the tinkering strategy in an effort to gain understanding into the meaning of programming constructs.

Struggling to understand the purpose of a variable in a piece of code that he had borrowed from an example, Jorge asks his programming partner, *"What do you think 'uptime' is?"*

```
function SetBallet(up_time) {
   DoTogether {
     RotateRightShoulder(0, 0, -170, up_time);
     RotateRightElbow(0, 0, -60, up_time);
     …
```

Both participants then turn their attention to the example code that contained the up_time variable. After isolating and testing the function that contained up_time, Jorge describes his strategy for deciphering its meaning, *"Ok, I was going to say we could, like, fiddle with the numbers to see if it does anything."*

Jorge then returns to the original code, replacing one of the up_time variable instances with the number 5, and testing a call to the SetBallet function.

```
main() {
  SetBallet(0.5);
  TwirlBallet(1);
}

function SetBallet(up_time) {
  DoTogether {
    RotateRightShoulder(0, 0, -170, 5);
    RotateRightElbow(0, 0, -60, up_time);
    …
```

His test produces an unbalanced movement, where the right shoulder rotates out of the step with the rest of the motion (more slowly, in this case). After executing the motion a couple of times, Jorge states his conclusions, *"Okay , I don't know what 'uptime' is but it definitely affects how they do stuff."*

Though Jorge's tinkering in this case did not lead to mastery of the argument and variable constructs, his testing actions and verbalizations demonstrate how tinkering with borrowed code is employed as a learning strategy to begin to resolve uncertainty.

4.1.4 De-coding

The participant who experimented with and mastered the most higher level constructs (function calls, movement constructs, loops and variables) was Sandra. During the analysis, we observed her spending long periods of time looking at specific pieces of code, often borrowed functions or sections. This seemingly passive action within the system is comparable to the "pauses" that Beckwith et al. observed in people who are successful debuggers [2]. After some consideration, we

interpreted Sandra's pauses as actions of de-coding, or unraveling the code to, in her words, "figure out how everything works."

Illustration 2: Sandra creates an acceleration effect using de-coding strategies

The following video transcript and coding excerpt demonstrates how Sandra employed this de-coding strategy, utilizing borrowed code as a learning resource.

Sandra wants to create a Twist dance motion, which she demonstrates physically with her own body intermittently while building her program. She creates a composite motion script (A) by borrowing the TwistHips function and calibrating the arguments (degrees and time), then creates an original loop construct (B).

```
function Main() {
    for (i=1; i<3; i++) {          (B)      (C)
        TwistHips(90, 2);  (A)
        TwistHips(-180, 2);
        TwistHips(90, 2);
        Wait(1);
    }                              (B)
}
```

Browsing the library, she asks her partner Amy, *"Do you remember which cheer it was where he was stomping and he, like, kept getting faster?"* They decide the motion was the Cheer3 function, which Sandra downloads and stares at for 30 seconds.

```
function Cheer3() {
    SetAttitudePose(0.5);                    (E)
    var movement_time;  (D)
    var wait_time;

    for (i=0; i<10; i++) {
        movement_time = 1.0 / i;
        wait_time = 0.5 / i;

        Wait(wait_time);              (I)
        StompRight(movement_time);
        Wait(wait_time);
    }
}
```

Researcher 1 asks, *"What are you thinking about?"*

Sandra: "*You just made him speed up and I'm trying to think about what did that, so (motioning with mouse over D) it's either the movement_time or the variable.*"

In this part of the excerpt, Sandra is trying to use acceleration in her own motion. She has seen this affect before, during Session 2's demo. Though she has previously used the loop construct successfully, she does not fully understand how variables and arguments work and therefore cannot yet create an acceleration effect on her own. Instead, she attempts to borrow code from the library that contains acceleration and modify it to accelerate her own motion. She locates and downloads the function from the library, then studies it intently. Her long periods of studying and subsequent coding actions were interpreted as *de-coding*.

Sandra then cuts her composite motion (C) and moves it to a workspace she uses to save code. She then copies/pastes all the code from Cheer3 (E) into her Main() and runs the script.

Sandra studies the motion and code. She then cuts/deletes (F) and replaces (G) with (A) – her previously created Twist motion.

```
function Main() {
    SetAttitudePose(0.5);          (F)      (E)
    var movement_time;
    var wait_time;

    for (i=0; i<10; i++) {
        movement_time = 1.0 / i;
        wait_time = 0.5 / i;

        Wait(wait_time);           (G)
        StompRight(movement_time);
        Wait(wait_time);
    }
}
```

Attempting to achieve the acceleration affect, Sandra has borrowed a section of code, then modified it to achieve her own creation by inserting her own code in the correct place.

```
function Main() {
    var movement_time;
    var wait_time;

    for (i=0; i<6; i++) {
        movement_time = 1.0 / i;
        wait_time = 0.5 / i;

        TwistHips(90, _);  (H)          (A)
        TwistHips(-180, 2);
        TwistHips(90, 2);
        Wait(1);
    }
}
```

Sandra runs the script. She watches it closely. This begins an extended period alternating between studying the Cheer3 function and her Main() code. Here, Researcher 1 prompts her to think aloud: "*So what are you thinking about?*"

Sandra: *"I don't know. It doesn't look like it's speeding up any. But that just could be my perception of it… I think it might have something to do with the numbers."* Here she scrolls down to look at the Cheer3 function.

Sandra: *"Ohhh. Huh, that's why."* She silently examines the code in Cheer3, then looks back to her Main function. She hovers her mouse cursor back and forth over (H) in Main then (I) in Cheer3. She then deletes the 2 in (H), leaves her cursor there in (H), and goes back to silently looking at code.

During this extended period of de-coding, through some querying by the researchers, we know that Sandra is trying to figure out what the arguments and variables mean and how they work. Instead of employing an active testing or tinkering strategy, by repeatedly plugging in new values and running them, she tries to understand first how the arguments interact as variables within Cheer3. The following exchange between Sandra and Researcher 1 sheds more light on how Sandra is using example code, a de-coding strategy, and her desire to create her own motion to understand new constructs.

Researcher 1: *"What are you thinking about?"*

Sandra: *"I'm thinking I need to write either move_time or wait_time in these brackets here, because that's what it looks like it did in the other one."*

Researcher 1: *"What do you think that would do?"*

Sandra: *"I think that would make it go faster. But, I also need, you know, I feel like I need that 90, -180 and 90* (italicized in H; by this she is referring to degrees in the 3D coordinate system) *to tell it where to go, obviously. Um. I feel like it needs a start time, but maybe that's what this 0.5 and 1.0* (reference to K) *are gonna do.*

So I might as well just try it instead of staring at it."

```
function Main() {
    var movement_time;
    var wait_time;

    for (i=0; i<6; i++) {
        movement_time = 1.0 / i;   (K)
        wait_time = 0.5 / i;

        TwistHips(90, movement time);
        TwistHips(-180, movement time);
        TwistHips(90, movement time);
        Wait(wait time);
    }
}
```

Sandra changes the numeric time values to the variable movement_time, and runs the script.

Before moving to modify and test a piece of code, Sandra tries to decipher what the code means, forming a hypothesis for how her changes will affect the Avatar's actions. This is the critical difference between the type of *tinkering* strategy that Jorge employed, and the *de-coding* strategy Sandra uses.

Taken as a whole, this illustration shows Sandra learning to use unfamiliar programming constructs through exploratory and creative engagement with the character's movements and the representations of the movements in the programming code. Attempting to mimic the acceleration action that she saw another avatar do, she looks for instances in existing programs that might emulate this. Extraction of the code and modification by way of testing to evaluate the consequential performance of her avatar eventually result in successful achievement of her goal, and some understanding—though partial—of the programmatic constructs that made it possible.

4.2 Conceptualizing & Parsing 3D Movement

Throughout the study, most of the observable pair interaction focused around conceptualizing movement in 3D. Contemplating, testing and discussing the 3D coordinate system and axis rotation of joints consumed large amounts of time for all participants, though this is not necessarily a negative outcome, as we will discuss later. Even within *creation* patterns, participants spent far more time testing and calibrating, changing numbers in an existing command or set of commands to bring the created action closer to the imagined action, than creating new motions. 3D was very hard for them:

Sandra: *"Especially with the rotating, that it's rotating around an axis so that if you rotate on the y it's going to move **like this**."* {Holding her arm straight down, and then rotating her shoulder joint (and elbow joint) around a vertical axis to twist her hand back and forth} ***The hand is going to move like that**… I honestly don't really understand it, that's probably why I'm having a hard time articulating it."*

This is one of many similar instances where a participant attempted to understand the 3D coordinate and rotation system through knowledge and observation of her own body enacting the desired movement. Five of our seven participants used their own bodies to mimic the 3D movements performed by the character, conceptualize movements, and understand rotation axes.

Illustration 3: Amy enacts desired avatar movement

The following excerpt shows a participant enacting movement with her own body to parse the movements she will need to generate to make her avatar do the same:

Amy: *"So we should make something pretty special, right?"* Amy clears out a large space in the textual window (below the main function) to use are her "work area," (M).

Amy: *"Oh I know what I want to do! I wonder if it could happen. Could I do like a snowman, like, when you do **a snow angel**"* (raising both arms up over her head in a snow angel motion.)

Before writing any code, Amy defines her function, "SnowAngel". Note: she introduces 2 syntax errors here that will not be addressed until much later.

Amy: *"I'm going to have to **rotate..."*** (lifting her arms up at her sides, rotating at her shoulder joint) ***"…the shoulder,"*** (continuing to move her arms up and down 3 more times).

Amy: *"Okay, so I'm going to have to rotate right and left at the same time. And, the, DoTogether..."*

```
Function SnowAngel();
    DoTogether {                    (M)
```

Amy: *"So it's **ro..."*** (raising her arms up and away again.)

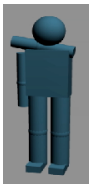
As Amy moves her arm to enact the motions she wants to create, she looks through available program code to see what might match to the enacted movement:

Amy turns to Researcher 1. Amy: *"So is RotateShoulder **like this**?"* (lifting her left arm straight out away from her at the side.) *"Or is it like **this**?"* (keeping her arm at her side, she rolls her left shoulder up, then back in its socket, then back to normal) *"…I don't know, I'll just try it out."*

When Amy tests the RotateShoulder motion, the Avatar's arm swings at the shoulder joint through its body – the opposite direction of her desired motion.

```
function Main() {
    RotateLeftShoulder(0, 0, -100, 0.7);
}

Function SnowAngel();
    DoTogether {                    (M)
```



Amy: *"Okay, so that's what I want it to do…"* So I have to remember..." Amy then spends about 2 minutes doing 3D conceptual work, drawing, verbalizing, and thinking. She changes the direction of the motion (by making it positive), then calibrates the motion code, repeating a cycle of changing the values, testing, then evaluating the effect until the Avatar motion matches her physically conceptualized motion. When she finds the right rotation, she moves the command into her SnowAngel function.

```
function Main() {
    RotateLeftShoulder(0, 0, 140, 0.7)
}

Function SnowAngel();                   (M)
    DoTogether {
        RotateLeftShoulder(0, 0, 140, 0.7);
```

In this illustration Amy enacts imagined movements for her avatar using her own body to parse movements to a smaller units of granularity that approach the programmatic level instruction that she sees in existing code. This connection between body and Avatar is key to understanding and manipulating her character in the 3D graphical world.

5. DISCUSSION

One objective is to understand how non-programmers approach a programming task that has the appeal of 3D animation as offered in Alice [6], but without 3D math hidden behind a veil of simplicity. Another is to theorize on how programmatic learning takes place in an end-user programming environment that has no tutorial or teacher.

5.1 Managing Multiple Representations

A way to explain the interactions of the participants with the environment is to view those interactions as a management of multiple representations of information. In this case, the information to be represented in different forms is 3D movement.

Our examination suggests that several representational states are at work, both visible and invisible. Four representational states of animation information explain the behaviors we observe. The *imagined action sequence of the avatar* is one representational state that we can say exists in the head of the participant [13]. The remaining representational states are external representations that exist in the world [8, 10, 13], and support individual and social cognition. The second representational state is the *physical enactment of a movement by the participant*. We saw this as having at least three purposes in this study: a) for communicating to one's partner the imagined, desired action sequence; b) to parse large, macro movements (such as rotating a shoulder as we saw in Illustration 2) into smaller movements that *map to a 3D coordinate system*; and c) to parse those same macro movements to smaller component movements that begin to *map to the granularity of programmatic constructs*.

A third representational state is that of the *programming code that creates movement for the avatar* (and within this, there are multiple representational states with respect to program modularity, the conventions of the programming language, and more, which here we will collapse into this single higher-order state). The fourth representational state is the *movement of the avatar*.

When testing, creating, modifying, borrowing, and de-coding behaviors are considered from this perspective, we can see that they involve the management of multiple representations of states of movement. Management of the representational states is not sequential in any kind of pre-determined sense; rather there is ongoing interplay between the employment and elaboration of representations.

From this vantage point, we may begin to see from which point users enter and tend to work within the representational system, resolve uncertainty, and maximize their creative skill. Some of the action employed to manage representational states is "pragmatic" while some is (also) what we would identify as "epistemic" [13]. Kirsh and Maglio explain that whereas pragmatic action changes the state of the world to bring it closer to some goal and reduce the cognitive burden on the actor by limiting the problem space, epistemic action captures a more interactional relationship between the state of the world and the cognition of the individual such that it changes the cognitive states of the individual simultaneously.

We can see the use of the representational states at work in the illustrations provided earlier. Amy shifted from an embodied representation of a desired action to an instructional (programmatic) one, using the Avatar's movements along the way to guide her creation/calibration. Sandra began with avatar movement she wanted to emulate, moved to searching for its programmatic representation, and then by moving back and forth between modifications on the programmatic representation and the Avatar's movement, achieved her goal and better understood aspects of programming. In this way, code-sharing, or code-borrowing and code-modification, in combination with graphical testing of the modifications, is an opportunity for representational transition that can result in learning.

Another instance of problem-solving that we saw at least once in each group illustrates how gaps in incomplete representations come to light when compared to other representations, and are opportunities for learning that in turn, elaborate incomplete representations. Programming the avatar requires that, to string or loop together borrowed program code into a realistic Avatar movement, the Avatar move back to a neutral position between transitions. Users tended to miss this requirement. These users would program the Avatar to execute what they thought was the next step in an action sequence, failing to realize that the start for the new action began at the finish point of the previous action, and that the Avatar (and its supporting code) did not "know" to resume a neutral stance on its own. This ended up producing contortions of the Avatar impossible for a real human body, which was puzzling (though also amusing) to users. It was at this point that a major insight was often made about the programmatic representation: that the program required more discrete and lower-level instructions than non-programmers realized.

Illustration 4: Discovering the representation gap

The following except shows Group B (Session 3) making this cognitive move towards a better understanding of what the programmatic representation required to produce the desired Avatar movement.

Jorge: *"Something, something didn't work."*

Mali: *"I think you need to put in there for him to reset."*

Jorge ignores the comment and changes the syntax of the loop he's testing, thinking that may have something to do with the strange result.

Jorge: *"He didn't do strange things (before)… and maybe he won't now." He runs his script again.*

Mali: *"No, he totally will, because he needs to go back to the normal position."*

Jorge: *"What do you mean?"*

Mali: *"Normal standing position, because he's doing all these motions from that, like, already in that pose."*

Jorge: *"…Ooohhh."*

Participants sometimes mapped the need-to-return-to-neutral discovery back to real 3D movement by enacting what the same move would require by the human body, and in so doing, we speculate, also fleshed out an impoverished representational state for human movement (which is so embodied that we do not normally parse it to finer granularities of movement). Mali and Jorge summed this up in the final debrief while talking about a video game:

Jorge: *"It makes a lot more sense now. Why they [computers] are so limited."*

Mali: "*In the first Sims game if you take the ladder off the swimming pool, they won't get out. They'll drown. It's ridiculous...*

Jorge: "*If you take out an important function, either he doesn't move at all, or you know...* (making a motion to contort his body

as in Illustration 3)... *it makes more sense now*."

5.2 A Gateway to Computational Thinking

These comments by Jorge and Mali, part of a larger conversation about how the two better understand now why the games they play behave as they do, indicate progress toward achieving "computational thinking," a skill that Wing believes has growing importance in today's world [17]. Other participants showed growth in computational thinking when realizing—because of the multiple representational states that they were able to work with in this environment—that looped actions have to return to the starting state between iterations, how to use heuristics to determine rotation planes, how to break complex human actions into precise components, etc.

Many of these breakthroughs in computational thinking related directly to manipulating the Avatar in unconcealed 3D. In many ways, our findings validate the early Alice design decisions to remove the demand for mathematical understanding of 3D [6]. The time spent dealing with these complexities was considerable, but does this consumption constitute a cognitive tax and negatively impact our overall goal of enabling novices to program? Or does it in fact fit within that goal? Though there is more work to do in answering these questions, based on this research we feel that the time spent understanding 3D movement represented simultaneous advancement toward programmatic understanding, and that the two worked in tandem, where actions taken upon the system of representations were inter-relational—epistemic—in nature [13].

6. FUTURE WORK

One future direction for the 3DAPS system is to embed the system within an existing online social network. Again, one strong piece of rationale for that design decision is the motivational component. Another justification is that such a move might broaden participation by bringing the end-user programming to a wider audience. However, this also opens up another connection between programmer and Avatar: a connection of identity. Future work will explore that connection, as well as the motivational impacts and pathways to learning in a socially-networked Avatar programming environment.

7. CONCLUSION

The 3DAPS system tested here and the larger social networked version that will eventually subsume it do not overtly "teach" programming and computational thinking. Instead, the hope is that such environments can provide motivation for and access to learning. In an end-user environment without a tutorial or a teacher, the information required for learning must be embedded in the system. This research shows how interaction and transition from coded representational states to graphically enacted motions, resulting from code-sharing and modification, can lead to programmatic learning. It also show how a 3D Avatar programming system can leverage existing, embodied knowledge to unravel less familiar knowledge—that of programmatic instruction—with 3D movement as language of translation between the representations.

8. ACKNOWLEDGMENTS

This research has been supported by the US National Science Foundation through the NSF Grant IIS-0546315 and an NSF Graduate Research Fellowship awarded to the first author.

9. REFERENCES

[1] Alac, M. Moving Android: On Social Robots and Body-in-Interaction. *Social Studies of Science* 39, 4 (2009), 37.

[2] Beckwith, L., C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrance, A. Blackwell, & C. Cook. Tinkering and gender in end-user programmers' debugging. In *Proc. SIGCHI 2006*, ACM Press, 231-240.

[3] Brandt, J., M. Dontcheva, M. Weskamp, & S.R. Klemmer, *Example-centric programming: integrating web search into the development environment*, in *Proceedings of the 28th international conference on Human factors in computing systems*. 2010, ACM: Atlanta, Georgia, USA.

[4] Brandt, J., P.J. Guo, J. Lewenstein, M. Dontcheva, & S.R. Klemmer, *Two studies of opportunistic programming: interleaving web foraging, learning, and writing code*, in *Proceedings of the 27th international conference on Human factors in computing systems*. 2009, ACM: Boston, MA.

[5] Bruckman, A. & M. Resnick. The MediaMOO Project: Constructionism and Professional Community. *Convergence* 1, 1 (1995), 17.

[6] Conway, M., S. Audia, T. Burnette, D. Cosgrove, & K. Christiansen. Alice: lessons learned from building a 3D system for novices. In *Proc. SIGCHI 2000*, ACM Press, 486-493.

[7] Crabtree, A. & T. Rodden. Hybrid ecologies: understanding cooperative interaction in emerging physical-digital environments. *Personal and Ubiquitous Computing* 12, 7 (2008), 481-493.

[8] Flor, N. & E. Hutchins, *Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Perfective Software Maintenance*, in *Empirical Studies of Programmers: Fourth Workshop*, J. Koenemann-Belliveau, T.G. Moher, and S.P. Robertson, Editors. 1991, Ablex: Norwood, NJ.

[9] Greenhalgh, C., A. French, P. Tennent, J. Humble, & A. Crabtree. From replaytool to digital replay system. In *Proc. 3rd Intl Conference on e-Social Science*, Citeseer.

[10] Hutchins, E. *Cognition in the Wild*. Cambridge, MA. MIT press 1995.

[11] Kelleher, C. & R. Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.* 37, 2 (2005), 83-137.

[12] Kelleher, C., R. Pausch, & S. Kiesler. Storytelling alice motivates middle school girls to learn computer programming. In *Proc. SIGCHI 2007*, ACM Press, 1455-64.

[13] Kirsh, D. & P. Maglio. On distinguishing epistemic from pragmatic action. *Cognitive Science: A Multidisciplinary Journal* 18, 4 (1994), 513-549.

[14] Papert, S. *Mindstorms: children, computers, and powerful ideas*. New York. Basic Books. viii, 230 p., 1980.

[15] Papert, S., *Situating constructionism*, in *Constructionism*, S. Papert & I. Harel, Editors. 1991, Ablex: Norwood, NJ, 1-11.

[16] Resnick, M., J. Maloney, A. Monroy-Hernendez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, & Y. Kafai. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60-67.

[17] Wing, J.M. Computational thinking. *Commun. ACM* 49, 3 (2006), 33-35.