Seamless Integration of Coding and Gameplay: Writing Code Without Knowing it

Stephen R. Foster
UCSD
srfoster@cs.ucsd.edu

William G. Griswold
UCSD
wgg@cs.ucsd.edu

Sorin Lerner
UCSD
lerner@cs.ucsd.edu

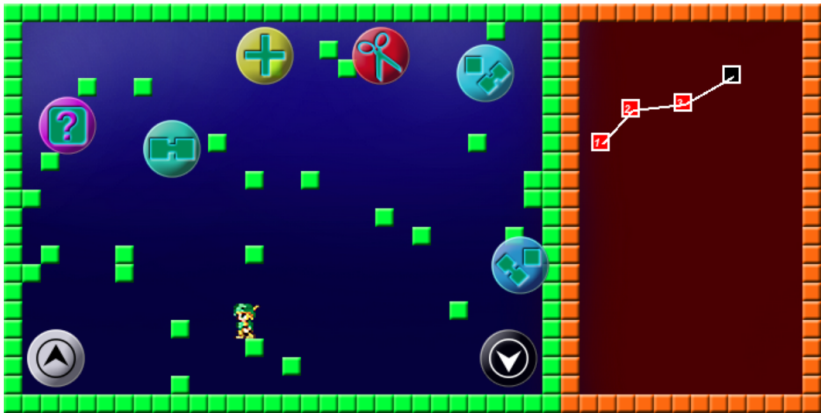

Figure 1. On its surface, The Orb Game is a 2D Mario-like game. The avatar (in the lower middle) can trigger various "orbs" throughout the environment. Doing so will transform the data displayed in the right-hand panel in various ways. In this game, users think they are playing a game and solving concrete puzzles. However, they are actually writing programs that operate on abstract inputs.

ABSTRACT

Numerous designers and researchers have called for seamless integration of education and play in educational games. In the domain of games that teach coding, seamless integration has not been achieved. We present a system (The Orb Game) to demonstrate an extreme level of integration: in which the coding is so seamlessly integrated that players do not realize they are coding. Our evaluation shows that the integration was successful: players felt that they were playing a game and did not realize they were programming. We also present a generalized framework called Programming by Gaming (PbG) to guide the design of other such games.

INTRODUCTION

A recent trend in educational game design and research goes by many names: seamless integration [6], immersive didactics [12], immersive learning [1], learning-gameplay integration [11], intrinsic integration [7], embedded learning [2], stealth learning [13], and avoiding "chocolate-covered broccoli" [10]. Though the words may be different, the sentiment is the same: Educational games should integrate learning and play, rather than artificially mashing the two together. A cogent empirical argument for why more integration is better is given by Habgood, who shows that tighter integration of content and gameplay correlates with higher motivation and better learning outcomes for players [7].

In this paper, we focus on educational games for teaching programming skills. Coding games are a domain in which a lack of integration can be easily seen in prior work – with the state of the art falling into two broad categories: 1) programming tools for building games (e.g. Project Spark, Scratch, and Alice) and 2) games in which programming is a game mechanic (e.g. Lightbot and CodeSpells). While these systems are engaging and educational in many ways, they do not qualify as (nor do they attempt to be) experiences that are *fully seamless*. It is still the case that:

Users can easily distinguish the coding portion of the experience from the rest.

A common theme in prior work is that there is a clear visual and interface difference between two distinct modes: a gaming mode and a coding mode. The user is made fully aware of the difference. For example, in Scratch, the interface is split between the code editing panel and the panel in which the code executes. In CodeSpells, there are distinct game modes: one in which the player navigates a 3D world and casts spells, and another in which the player writes code to create spells.

The main contribution of this paper is a game design that seamlessly merges coding and gaming into a single mechanic. To achieve this game design, we leverage techniques from

"programming by demonstration" (PbD), a kind of end-user-programming in which users demonstrate actions on concrete values in order to construct algorithms. In addition to leveraging off-the-shelf PbD techniques, we also address many of the challenges of mapping PbD onto a gaming interface. While there is a long line of research on programming by demonstration, its use in a gaming environment appears to be novel, and there are unique challenges and opportunities in this domain. For our purposes, programming by demonstration enables the player to act on objects in a gaming environment, while simultaneously demonstrating to the system the steps that the program should take. More specifically, by mapping familiar platformer game mechanics onto various data display and transformation operations, our system allows users to demonstrate various transformations through familiar gameplay, without having to go into a separate coding interface. This leads us to a general technique we call Programming by Gaming (PbG) as well as a particular instantiation of PbG in a game called The Orb Game.

The Orb Game is designed to make players feel that they are playing a Mario-like platform game, while they are actually writing algorithms. Because "fully seamless" is an empirically testable metric, we present an evaluation that shows that players did not realize they were actually writing algorithms when they played The Orb Game. This suggests the PbG approach can be used to design other fully seamless coding/gaming experiences – i.e., ones in which users cannot distinguish the coding from the rest.

FULLY SEAMLESS DESIGN

Programming by Gaming

For coding games, the goal of being fully seamless decomposes nicely into two form/function subgoals:

Game-like form factor. The "form-factor" of the interface should resemble a game, not a programming language. It should look and feel like a game.

Language-like functionality. But the system needs to *also* function like a programming language – i.e. can solve abstract computational problems.

At the highest level, a PbG system maps code-writing operations to in-game actions – thus obtaining something that looks like a game but functions like a programming language. More specifically, though, PbG borrows from the overarching philosophy of PbD (programming by demonstration), mapping the user actions to *immediate* effects on concrete values, while generating a more abstract algorithm in the background. PbG further constrains PbD by requiring that the user's actions must be framed as gameplay actions. We demonstrate a PbG design by introducing The Orb Game, which maps in-game platformer genre mechanics onto various functional programming language operations.

The Orb Game

To understand The Orb Game, let's look at a usage scenario. Suppose that Bob sits down to play The Orb Game. Let's suppose that Bob has some test cases. There are various possibilities here: the test cases came from Bob's teacher; they came from a busy professional programmer who wants to save time by crowdsource the writing of a subroutine to a worker on Mechanical Turk; they came from an automated-tutoring system who is serving up a pedagogically relevant problem; or Bob wrote the test cases himself to solve some computation problem (and for whatever reason, Bob is playing The Orb Game instead of a more traditional coding interface like Python or Excel to derive his answer).

For example, if Bob is supposed to to write an algorithm that sums up a list of numbers, the test cases might be: "[1,2,3] to 6", "[2,3] to 5", "[3] to 3", and "[3,4,5,6] to 18". Perhaps whomever provided the test cases also provides a more human-readable specification like, "Add up the list of numbers" (though this is not strictly necessary).

When Bob begins the game, these test cases have already been provided to his system, so he finds himself confronted with the interface shown in Figure 1. Notice that the numbers on the right-hand panel match one of the test cases.

We will analyse each part of The Orb Game as both a programming language construct and as a game-mechanic (e.g. as PbG mappings):

Mission. The directive to "Add up the list of numbers", though not shown in the interface in Figure 1, is common in both games and programming. So the mapping is quite natural.

- *Game-like form factor*. Games often contain implicit directives – i.e. don't die, don't run out of time, collect all the coins – and explicit directives – i.g. "infiltrate the enemy based and retrieve the documents." Explicit directives are known as "missions" or "quests", depending on the genre of the game.

- *Language-like functionality*. For programmers, such directives are known as "specifications" or "requirements".

Inventory. The inventory, the right-hand panel of the interface depicted in Figure 1.

- *Game-like form factor*. The inventory represents the items that the player is carrying. This is a common mechanic found in roleplaying games, where players routinely find, pick up, and carry in-game equipment in their inventory. This construct can be assumed to be familiar to players of many popular games, such as *World of Warcraft* and *Skyrim*.

- *Language-like functionality*. Insofar as the game is also a programming language, the inventory contains representations of the data on which the program is operating. It is essentially the "heap". In Figure 1, the inventory contains a linked list, which in the context of our Bob example was auto-generated based on the test case.

Avatar. The avatar is the small character standing on a green block in the center of figure 1.

- *Game-like form factor*. Avatars are common in almost all games and represents the player's in-game persona. This particular avatar inhabits a 2D world and can jump from

platform to platform – a mechanic found in classic so-called "platformer" games like *Mario* and *Prince of Persia*.

- *Language-like functionality*. The sequence of avatar actions serve to define the program's control flow. As the player manipulates the avatar, the actions are recorded and can be played back at runtime.

Activatable Entities. The avatar can interact with the colored orbs shown in Figure 1.

- *Game-like form factor*. Many games contain things that the avatar can interact with. *Mario* has special blocks with question marks on them that produce items of interest when the avatar touches them. Other manifestations are pressure plates, traps, buttons, switches, treasure chests, and doors.

- *Language-like functionality*. As a programming language, these orbs represent primitive functions that can operate on the data in the inventory. The one with a plus sign can add two inventory items together. The one with the scissors can cut the first item off of a list. As a group, such operations can be thought of as an API – a collection of related functions.

Bob is familiar with Mario-style platformers, so he takes control of his avatar and begins to navigate The Orb Game. First, he enters the white orb in the lower lefthand corner of the screen – the "return" action. Insofar as the game is a programming language, this represents returning from the main function with whatever is contained in the avatar's inventory. Because the avatar is carrying the same list as when the game began, Bob has written the identity function. But because "[1,2,3] to [1,2,3]" wasn't one of the test cases – the game informs Bob that he has not solved the puzzle, that he should only exit the level when he has a "6" selected. This is Bob's first incorrect solution.

Now Bob explores for some time and comes up with the following (also incorrect) solution. First, Bob causes his avatar to touch the red orb with the scissors icon. This action represents the "pop" function. This causes the first element of the linked list [1,2,3] to become separated from the list (a destructive action that both returns the first element from the list and removes the first element from the list). See Figure 2.2.

Although Bob has performed a concrete action on a concrete list, the system interprets the action abstractly. In other words, Bob has written the following code (though he doesn't know it):

```
a = pop(input)
```

(The variable *input* represents the input to the sum function, which Bob is unknowingly writing.)

Bob selects the popped element (the number 1) and carries it to the yellow orb – the "add" action. Code:

```
a = pop(input)
b = add(a, ...)
```

The ellipsis above (...) represents that the add function has only been partially applied. Now Bob selects the list [2,3] and

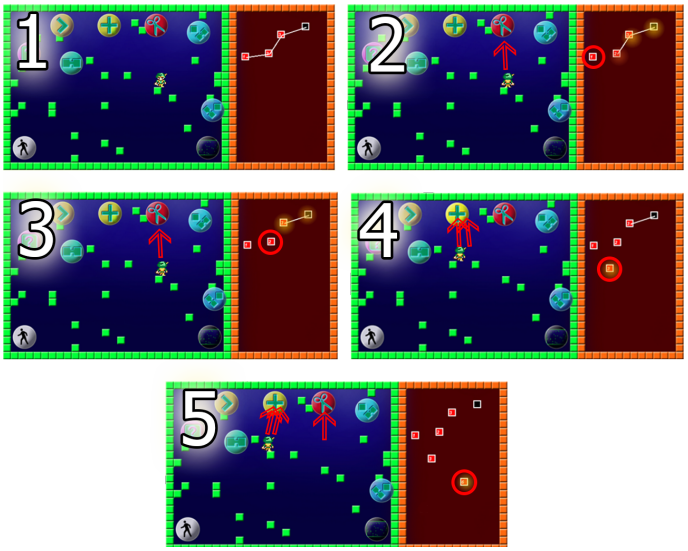

Figure 2. 1) Bob begins with the list [1,2,3]. 2) Bob triggers the pop action, popping the first element from the list. 3) Bob pops the second element from the list. 4) Bob triggers the add action twice, adding 1 and 2, producing 3. Finally, 5) Bob does another pop and another add, producing a 6.

touches the red orb again – the "pop" function. The element 2 becomes separated from the list. See Figure 2.3. Code:

```
a = pop(input)
b = add(a, ...)
c = pop(input)
```

Now, Bob selects the 2 item and carries it to the yellow *add* orb, which produces the number 3 (now that it has received both necessary inputs to perform addition). See Figure 2.4. Code:

```
a = pop(input)
c = pop(input)
b = add(a, c)
```

Notice that the second and third lines have been swapped. Because the *c* variable has been used to complete the *add* function, the compiler is able to enforce the correct ordering.

Bob does one more application of *pop* (producing another 3). He proceeds to add this 3 to his previously produced 3. See Figure 2.5. The *add* box now produces 6. See Figure 2.5. Bob selects the 6 and exists the level again via the "return" orb. After all of these concrete actions, we have the following abstract code:

```
a = pop(input)
c = pop(input)
b = add(a, c)
d = pop(input)
e = add(b,d)
return e
```

The game produces a congratulatory message because Bob has found a correct concrete solution for this concrete input. However, his solution is not very general – a fact that is revealed to him when the game replays his sequence of actions

before his eyes on another one of the test cases ("[3,4,5,6] to 18"). He sees his avatar go through the same process as before, but exiting the level with the number 12. The game informs Bob that he needs to come up with a single solution that works for all inputs.

Writetime vs Runtime. This brings us to one feature of programming languages that is not found in most games.

- *Game-like form factor*. There are some games that involve a kind of record/replay mechanic. For example, the game *Braid* involves manipulation of time, and the players actions can be rewound and replayed. In the game *The Incredible Machine*, the player builds a virtual Rube Goldberg contraption, then presses "Play" and watches it run. In the popular game of *Starcraft*, one gives a series of orders to various troops and then watches those troops perform those operations.
- *Language-like functionality*. All language interfaces have a writetime and a runtime. The distinction between the two blurs in so-called "reactive" interfaces like Excel – where the modification of one cell can cause a cascade of changes across other cells – and in programming by demonstration systems, where concrete actions are automatically performed on concrete objects. However, a distinct runtime is still necessary when testing the same algorithm on a different concrete object – as is the case with Bob's process.

Conditionals and Recursion: The Big Problems

To solve this puzzle for all possible inputs, Bob needs a language that has either loops or recursion. In either case, though, the idea is that a sufficiently powerful language needs to be able (at runtime) to return back to a previous line of code. At writetime, the programmer needs to be able to specify when such returns ought to occur. Such returns need to be conditioned upon the data (so that loops can terminate). We chose to implement recursion because of our background as functional programmers.

Up to this point, Bob has performed actions that were executed immediately. When he popped an element off of the list, he saw the element become separated from the list in his inventory immediately. If he were to add two numbers together, he would see the result appear in his inventory immediately. In other words, although he is writing code, the system is also running his code as he writes it.

Let's assume suppose Bob derives the following recursive solution. Bob pops the first item off the input list [1,2,3] (as he did before). See Figure 3.2. He selects the rest of the list [2,3] and activates the black orb located at the bottom right of the game world. See Figure 3.3. This orb represents making a recursive call to the current function[1]. Ideally, the environment would now place into Bob's inventory the result of the recursive call. This is impossible (in general), however, because the function Bob is writing is not yet complete. But the recursive call is being performed on the list [2,3], which happens to be another one of the test cases provided before Bob began. So the system knows the answer even though the algorithm is not complete. This allows the number 5 to be placed into Bob's inventory.

He then takes the number 5 and uses the addition orb to add the 5 to the number 1 – which he popped off earlier. This produces the number 6. He selects the 6 and exits the level by touching the white return orb. Here's the code he unknowingly generated:

```
a = pop(input)
b = sum(input)
c = add(a, b)
return c
```

The Orb Game will then switch to runtime, replaying the avatar's actions on the same input. Up until the point where the avatar touches the recursion orb, the replay will be straightforward. But when the avatar touches the recursion orb while the list [2,3] is selected, a new instance of the game will spawn (a new "stackframe" in programming language terms) on top of the current instance. The replay will begin anew with the list [2,3] in the inventory. The avatar will pop off the 2, select the [3], and touch the recursion orb – spawning yet another instance of the game (another stackframe). One more replay, and the avatar touches the recursion orb with an empty list. This replay will fail on the pop action, so the game revers back to writetime, allowing the player to continue playing from that point on – with the empty list in the inventory. The fact that the execution failed on the empty list allows the system to construct the following code:

```
if(input == [])
    ...
else
    a = pop(input)
    b = sum(input)
    c = add(a, b)
    return c
```

The correct thing to do in this base case is to activate the "define constant" orb (the purple question mark orb shown in the left on figure **??**), which will prompt Bob for input. He inputs the number 0, which is immediately placed into his inventory. He then selects the 0 and returns, completing the second branch of the conditional.

```
if(input == [])
    a = 0
    return a
else
    a = pop(input)
    b = sum(input)
    c = add(a, b)
    return c
```

Now the game switches back to runtime and continues by popping off the topmost game instance (stackframe). The zero from that instance is placed into the inventory in the instance beneath. The replay now continues, adding the result of that recursive call to the item popped from the list yielding a 3 (3 + 0 = 3). Still replaying Bob's actions from earlier,

[1]We chose recursion instead of loops because of our background as functional programmers. Some kind of looping mechanic would also have been completely viable.

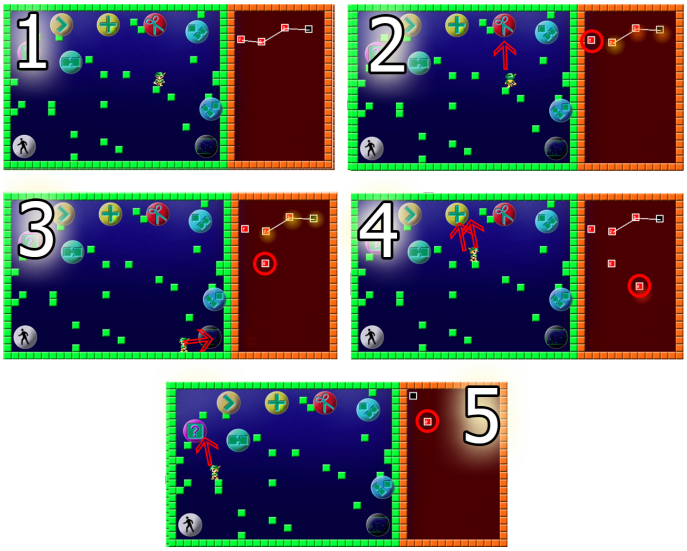

Figure 3. 1) Bob begins with the list [1,2,3]. 2) Bob triggers the pop action, popping the first element from the list. 3) Bob triggers the recursion orb; the oracle returns 5. 4) Bob triggers the add action twice, adding 1 and 5, producing 6, which he returns. Finally, 5) Bob must solve the base case, which he does by activating the define constant orb, getting a 0, and returning.

this new 3 is selected and the avatar touches the return orb – popping off another game instance (stackframe). The returned 3 is carried into the instance beneath, where it is added to the 2, yielding a 5 to be returned. And so on, until a 6 is returned from the bottom-most game instance. This matches the expected return value for the test case. So the system now attempts to try the same sequence of actions on the other test cases. See 4 for an image of the stacked game instances.

As we can see, the point of the visualized program execution is two-fold:

- If Bob has correctly solved the puzzle, the replay gives Bob an explanation for why his answer is right, as well as (hopefully) some gratification in seeing his solution correctly handle all the test cases.
- If Bob has not correctly solved the puzzle, the replay is analogous to a debugger – it visualizes every step of the program execution at a speed conducive to human comprehension, allowing Bob to see where his solution breaks down.

In this example, Bob has succeeded in producing the correct general solution. Of course, we have elided much of Bob's learning curve. In our Evaluation section, we tackle the foundational questions in this line of research: Does the experience feel like a game? And can non-coders really produce correct general solutions to problems by playing this game?

EVALUATION

The goal of our experimental evaluation is to understand the extent to which our PbG approach, and more specifically its instantiation in The Orb Game, provides fully seamless integration of coding and gameplay – where "fully seamless" means that *players think they are playing game, and do not realize they are coding*. To this end, we recruited 12 subjects from non-STEM majors at a local university and had them solve various textbook-style problems dealing with linked-list processing. We prescreened for subjects with no programming experience. On further investigation, three of the subjects turned out to have experience with programming, leaving us with 9 subjects (1 man and 8 women). We did not tell the users that the system was called "The Orb Game" – so as to avoid priming them to think of it as a game.

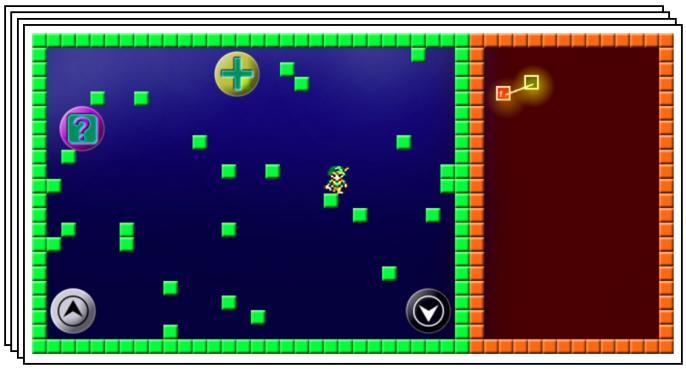

Figure 4. The runtime stack visualized as an actual stack of games during execution. When the avatar triggers the return orb in the lower left, the top-most level will be removed, and the currently selected items in the inventory (the list [1]) will be returned to the next game in the stack.

We designed two sets of programming tasks: a set of training tasks and a set of benchmark tasks. The training tasks were intended to familiarize users with the interface – i.e. to mimic the "training levels" often found in commercial games. The benchmark tasks were intended to assess whether (after completing the training tasks) the users could solve more complex problems without assistance. The training tasks were divided into two groups: basic and advanced. The basic tasks were as follows and can be categorized according to the various API calls (represented by the orbs) necessary to complete the task:

- *Add/Return*. Add two numbers together and return the result.
- *Pop/Add/Return*. Pop two numbers off a list, add them together, and return the result.
- *Pop/Add/Return*. Pop three numbers off a list, add them together, and return the result.
- *Pop/Concat/Return*. Pop the first two numbers off a list, concatenate the first one back on, and return the result.
- *Pop/Max/Return*. Pop the first two numbers off of a list, return the larger of the two.
- *Pop/Constant/Add/Return*. Pop the first number off of a list, add 6 to it, and return the result.

The advanced training tasks were designed to demonstrate recursion and the importance of producing a general solution (i.e. one that works on all inputs):

- *Max*. Return the maximum element in the list.
- *Sum*. Return the sum of all elements in a list.

- *Even/Odd*. Return 1 if there are an odd number of items in the list and 0 otherwise.

The benchmark tasks were as follows:

- *Reverse*. Reverse the order of list items. We designed this benchmark to be isomorphic to *sum* and *max* – i.e. the correct answer is to pop the first element off the input (let's call the result F), perform a recursive call (let's call the result R), and return the result of a binary operation on F and R. It is different, though, in that the return type is a list instead of an integer.

- *Map +5*. Return the input list, but with each element incremented by 5. We chose this benchmark because it is involves several different operations – obtaining a constant, addition, recursion, and concatenation. The general case is, therefore, quite complex (more complex than any of the training tasks). The base case, however, is simple.

- *Return last*. Given a list as input, return the last element. We chose this benchmark to assess subjects' performance when the game's automatically-generated base case is not correct. (The correct base case handles a list with a single element and returns that element.)

All of the advanced training tasks and the performance benchmarks are programs that operate on a single input – a linked list. We provided test cases such that the oracle would return a correct answer for a recursive call on any sublist of the input.

For each subject, we conducted a 90 minute session structured as follows:

Basic training. The first 30 minutes was spent asking the subject to perform each of the basic training tasks. The subject was encouraged to ask questions.

Advanced training. In the second 30-minute period, the researcher conducting the experiment spent 10 minutes on each of the 3 advanced training tasks. In each of these 10-minute segments, the researcher first discussed the problem with the subject. Then the user was permitted to take control of the avatar and attempt a solution. Asking questions was permitted and encouraged. Then the researcher took control of the avatar and demonstrated the correct solution, pausing for frequent Socratic interludes – i.e. asking the subject "What do you think will happen when I do this?" The researcher made a point to articulate a common 4-step pattern in all three solutions: 1) Reduce the input, 2) use the black orb (the recursion orb) to obtain a solution for the reduced input, 3) figure out how to use this solution to obtain a solution for the original input, and 4) solve the base case.

Performance Benchmarks. In the final 30-minute period, the subject was instructed not to ask questions, unless to clarify the problem statements. The subject was given 10 minutes to complete each of the performance benchmarks. The subject was allowed to make as many attempts as time permitted. The 10-minute segment was ended early in the event that the subject obtained the correct answer in less than 10 minutes. The researcher recorded the time it took the subject to obtain a correct result, as well as the number of incorrect attempts made beforehand.

Reverse	*Map +5*	*Last*
3 min, 3 tries	**3 min, 4 tries**	**4 min, 2 tries**
6 min, 3 tries	**8 min, 2 tries**	**8 min, 3 tries**
8 min, 4 try	**3 min, 2 tries**	Fail, 2 tries
3 min, 1 try	**7 min, 2 tries**	Fail, 3 tries
3 min, 1 try	**10 min, 4 tries**	Fail, 2 tries
2 min, 1 try	**10 min, 6 tries**	Fail, 2 tries
2 min, 1 try	**2 min, 1 try**	Fail, 2 tries
2 min, 3 tries	Fail, 4 tries	Fail, 2 tries
5 min, 2 try	Fail 6, tries	Fail, 3 tries

Figure 5. Seven of the subjects completed at least 2 benchmarks. All subjects completed at least 1 benchmark. For those who completed the benchmarks, the average times for completion were 3.8 minutes for the first, 6.1 minutes for the second, and 6 minutes for the third.

The remaining time was spent on a short semi-structured exit interview. Questions asked included: What did you find difficult? What was the most difficult puzzle? What if anything would make the game more fun? What (in your own words) does the black orb do?

Results

The results of each subject's performance on each benchmark are contained in the table in Figure 5. All of the subjects solved at least one of the three puzzles.

The most solved benchmark was the *reverse* benchmark – which all subjects were able to correctly complete. The least-solved benchmark was the *last* benchmark, which was correctly completed by two subjects.

We believe the *reverse* benchmark was relatively simple because it was isomorphic to the *max* and *sum* training tasks. The *last* benchmark was difficult due to the fact that the automatically-generated base case was not the correct base case. All subjects did correctly solve the general case for the *last* benchmark, though. The two subjects who correctly saw the need for a different base case first gave an incorrect solution, only correcting their solution after watching the in-game execution.

Inventory. All users made use of the fact that inventory items could be dragged around. They used this feature to organize their data values into various groups while performing tasks. Every subject used this feature on every benchmark task, indicating that they found it useful.

In-game execution. The runtime environment was valuable in all tasks in which players were not able to solve the puzzle correctly the first time (see the cells in Figure 5 in which the subject had to try more than once). In these tasks, players would play, then watch the replay, then repeat the process. Because these cycles resulted in a correct solution on so many of the tasks, we can conclude that the game's runtime environment is critical to the code writing process.

We observed two pervasive incorrect strategies that pertain to the game's use of concrete values. Subjects had some initial

trouble grasping that they were supposed to produce a general solution.

Deconstruct and reconstruct. All subjects, on at least one benchmark, utilized a "deconstruct and reconstruct" strategy. In other words, they would pop all items off of the input list, perform the intended computation (e.g. find the sum), and return the result – avoiding a recursive call altogether, and ultimately producing a non-general solution (one that doesn't work on a list with a different number of elements). Thus, in spite of the prior training, subjects retained a strong preference for concrete solutions. This indicates that our game's next iteration should contain multiple training levels to help acclimate the player to the process of producing a general solution.

Building a constant. More than half of the subjects, during the *map+5* task, attempted to produce the constant 5 by popping numbers off the input list and adding them together to construct a 5. This was in spite of the fact that an orb for producing desired constants was provided. We suspect that this unproductive strategy would not have arisen if we had provided more than one basic training task involving constants. We plan to include this in our game's next iteration.

For both of these strategies, the existence of the game's runtime visualizations helped students recognize and get past their initial confusions, demonstrating the value of this feature.

Difficulties and confusions. During the interview, subjects were asked what the most difficult puzzle was – to which all but one named the *even/odd* puzzle. This was interesting because this puzzle was one of the training tasks, which the subject and the researcher solved in collaboration. The difficulty was likely because the task was the only task that required 2 different base cases: 1) a base case in which the input is the empty list and 2) a base case in which there is only one item in the list. (The general case simply pops twice and returns the result of a recursive call). This indicates that perhaps the game should provide more support and training with regard to base cases.

What does the black orb do? Two of the subjects said that it reminded them of the movie *Inception*, whose plot features dream worlds within dream worlds – which is indeed a manifestation of recursion. Other perceptions of the black orb focused on its functionality as an oracle: one subject said "It lets you cheat", and an other said "It's like when Dumbledore gave Harry Potter the snitch and it took him forever to figure out what it meant." The other subjects simply described its mechanics in a more literal fashion.

Seamless integration. Most importantly, when asked during the interview what sorts of things the system reminded them of, all subjects mentioned some kind of game (Mario being the most common). "Solving a puzzle" was also common. Only one subject mentioned that it was like "getting the computer to do stuff for you" – which perhaps indicates a more code-like perception than a game-like one.

We can conclude from the above results that: 1) using familiar game mechanics (e.g. from platformer-type games) did indeed make people feel that they were playing a platformer game; 2) difficulties were presented by the game's recursive nature, making the experience also feel like a puzzle game rather than the intended platform game; and 3) the record/replay (writetime/runtime) mechanic greatly assisted with producing general solutions, while having no negative effects on perceptions of the system as a game.

Threats to Validity

A potential threat to our conclusion that The Orb Game was not recognizable as a coding interface is that, perhaps by recruiting non-coders, we recruited people who would have lacked the ability to recognize that they were coding even if they were presented with an obvious coding interface – like Scratch or Python. The reason we think this is *not* the case, though, is two-fold:

- We routinely run studies on non-coders (calling for participants with "no coding experience"). These participants, though they have never coded, overwhelmingly tend to have an accurate general understanding of coding. They know basically what coding is even if they haven't done it. And they do recognize code (even Scratch or Alice code) when they see it.
- Much research has been done on the adoption of perceptions of coding by young people, showing that they form perceptions about coding (and whether or not they could ever be a coder) at an early age, often before high school [8]. This research suggests that the basic ideas of coding are something that even non-coders are exposed to from an early age.

Furthermore, our subjects in this study expressed surprise and even skepticism when told, at the end, that they were writing code while playing the game.

RELATED WORK AND DISCUSSION

Programming by demonstration. One of our design decisions was to allow the user to manipulate concrete values instead of abstract values – a technique used in programming by demonstration systems. Early systems of this sort were were Pygmalion [3] and Tinker [9]. The motivation for these systems is the same: Cognitively speaking, human beings seem to understand concrete values (like 7) better than abstract ones (like n). And when writing an algorithm like, say, the factorial function, it can be easier for a person to demonstrate how to obtain the factorial of 7 than it is to write the more general factorial function. Often, the system can produce the more general function from the demonstration. Although some programming by demonstration systems are intended to be educational, we know of no programming by demonstration systems that is designed to disguise itself as a game.

Coding education systems. Although gamification is not found in programming by demonstration systems, it is commonly found in tools for educating novices about programming. The Logo language began a long tradition of building tools to make coding more accessible to those who don't already know how to code. Game programming was an early

domain for Logo. Today, the Alice and Scratch environments are commonly used to code games in introductory programming classes. Other environments for children and/or novices revolve around games – e.g. Kodu, CompetitiveSourcery [6], and CodeSpells [5]. Many of these are tools for making games, but not games themselves. However, even the examples that do try to integrate coding into gameplay do so by relegating coding to a discrete interface mode.

By leveraging PbG (PbD plus mappings from coding to gameplay operations), our system allows coding and play to occur simultaneously. During writetime, gameplay and code writing are seamlessly integrated. And during runtime, game*re*play and code execution are seamlessly integrated. And by "seamlessly integrated", we mean that they are literally the same thing. This is a level of seamless integration between coding and gameplay that has not been achieved by previous coding tools for novices.

Situated as it is between two research traditions ("programming by demonstration tools" and "novice coding tools") our system has at least two possible futures. The system can be made more useful and/or more educational.

More useful. There is a term for games that have useful gameplay: they are "games with a purpose", an idea pioneered by Louis von Ahn [14]. Pipe Jam [4], for example, is a puzzle game that tricks players into writing correctness proofs for programs. The point of such games is to crowdsource useful work to players. Given the results of our study (wherein certain tasks were found to be too difficult), some training levels are needed. Furthermore, the game would need to be extend to allow players to manipulate more powerful structures – e.g. objects.

More educational. We did not attempt to make The Orb Game particularly fun – and many of our subjects suggested that they would have liked to see some traditional obstacles like traps or multiplayer. That said, the history of the platformer game genre is rife with mechanisms that could trivially be included in our prototype. After making the game more fun, it will also be important to perform logitudinal studies to discover whether the game (though fun) has real educational value.

CONCLUSION

In education games research, seamless integration has been empirically validated and called for repeatedly. We made it our goal to integrate coding and gameplay so seamlessly that players would not know they were writing code (a benchmark we term "fully seamless"). The technical challenges of the domain (i.e. mapping gameplay actions to code, representing runtime and writetime, incorporating recursion) can be surmounted while preserving seamlessness. We did this by building a PbG system, which *merges gameplay with coding in a programming by demonstration system that maps concrete player actions to code production.* More specifically: 1) Simple data transformations are mapped to platformer game mechanics that have immediate effects on the data; 2) the sequence of actions is generalized into code in the background; 3) The writetime/runtime distinction is mapped to the play/replay game mechanic; 3) Recursion is represented at writetime as an orb that returns an inventory value, and at runtime as a stack of game replays; 4) Conditionals are represented as pattern matches on the inventory contents at the beginning of a recursive call.

We now know that it is possible to get non-coders to write algorithms without knowing it. Furthermore, such algorithms work on all inputs, not just the test cases that the player is given.

REFERENCES

1. Adamo-Villani, N., and Wright, K. Smile: An immersive learning game for deaf and hearing children. In *ACM SIGGRAPH 2007 Educators Program*, SIGGRAPH '07, ACM (New York, NY, USA, 2007).
2. Bellotti, F., Berta, R., Gloria, A. D., and Primavera, L. Enhancing the educational value of video games. *Comput. Entertain. 7*, 2 (June 2009), 23:1–23:18.
3. Cypher, A., Halbert, D. C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B. A., and Turransky, A., Eds. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993.
4. Dietl, W., Dietzel, S., Ernst, M. D., Mote, N., Walker, B., Cooper, S., Pavlik, T., and Popović, Z. Verification games: making verification fun. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, ACM (New York, NY, USA, 2012), 42–49.
5. Esper, S., Foster, S. R., and Griswold, W. G. On the nature of fires and how to spark them when you're not there. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, ACM (New York, NY, USA, 2013), 305–310.
6. Foster, S. R., Esper, S., and Griswold, W. G. From competition to metacognition: Designing diverse, sustainable educational games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, ACM (New York, NY, USA, 2013), 99–108.
7. Habgood, M. P. J., and Ainsworth, S. E. Motivating children to learn effectively: Exploring the value of intrinsic integration in educational games. *Journal of the Learning Sciences 20*, 2 (2011), 169–206.
8. Jacobs, J. E., and Simpkins, S. D. *Leaks in the pipeline to math, science, and technology careers / Janis E. Jacobs, Sandra D. Simpkins, editors*. Jossey-Bass San Francisco, 2005.
9. Lieberman, H., and Hewitt, C. A session with tinker: Interleaving program testing with program design. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, ACM (New York, NY, USA, 1980), 90–99.
10. Linehan, C., Kirman, B., Lawson, S., and Chan, G. Practical, appropriate, empirically-validated guidelines for designing educational games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, ACM (New York, NY, USA, 2011), 1979–1988.
11. Maciuszek, D., and Martens, A. A reference architecture for game-based intelligent tutoring. In *Handbook of Research on Improving Learning and Motivation through Educational Games: Multidisciplinary Approaches*, P. Felicia, Ed. IGI Global, Hershey, PA, 2011, ch. 31, 658–682.
12. Maciuszek, D., Weicht, M., and Martens, A. Seamless integration of game and learning using modeling and simulation. In *Proceedings of the Winter Simulation Conference*, WSC '12, Winter Simulation Conference (2012), 143:1–143:10.
13. Prensky, M. *Digital Game-Based Learning*. McGraw-Hill Pub. Co., 2004.
14. von Ahn, L., and Dabbish, L. Labeling images with a computer game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, ACM (New York, NY, USA, 2004), 319–326.

This work was supported in part by NSF grant CCF 1423517.