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

Stephen R. Foster
UC San Diego
9500 Gilman Dr. # 0404
La Jolla, CA 92093-0404
srfoster@cs.ucsd.edu

Sorin Lerner
UC San Diego
9500 Gilman Dr. # 0404
La Jolla, CA 92093-0404
lerner@cs.ucsd.edu

William G. Griswold
UC San Diego
9500 Gilman Dr. # 0404
La Jolla, CA 92093-0404
wgg@cs.ucsd.edu

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 doing it.

1. INTRODUCTION

A recent trend in educational game design and research goes by many names: seamless integration [3], immersive didactics [7], immersive learning [1], learning-gameplay integration [6], intrinsic integration [4], embedded learning [2], stealth learning [8], and avoiding "chocolate-covered broccoli" [5]. 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 [4].

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 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.*

The main contribution of this paper is a game design that seamlessly merges coding and gaming into a single set of mechanics. 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. For our purposes, PbD enables the player to act on objects in a gaming environment, while simultaneously demonstrating to the system the steps that the program should take. 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. 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.

Proceedings of the 10th International Conference on the Foundations of Digital Games (FDG 2015), June 22-25, 2015, Pacific Grove, CA, USA.
ISBN 978-0-9913982-4-9.

2. FULLY SEAMLESS DESIGN: THE ORB GAME

To understand The Orb Game, let's look at a usage scenario. Suppose 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 these are Bob's own test-cases, or they were provided by someone else – like a teacher. Either way, Bob 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.

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. 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. For programmers, such directives are known as "specifications" or "requirements".

Inventory. The inventory, the right-hand panel of the interface depicted in Figure 1.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*. 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. 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*. 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. 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. 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. Now Bob selects the list [2,3] and touches the red orb again – the "pop" function. The element 2 becomes separated from the list. See Figure 2.3. Then, 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.

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
```

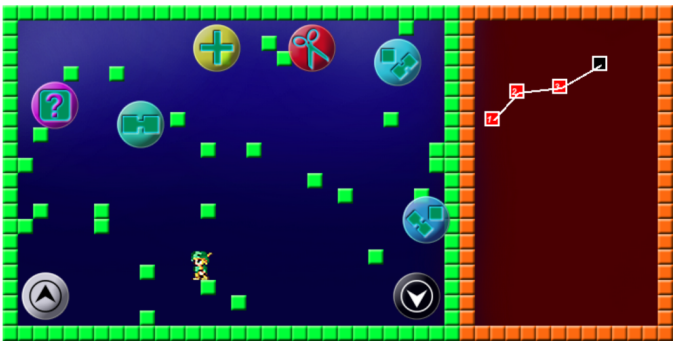

Figure 1: On its surface, The Orb Game is a 2D Mario-like game. The avatar (in the lower middle) can trigger various "orbs" throughout the environment. Doing so will transform the data displayed in the right-hand panel in various ways. In this game, users think they are playing a game and solving concrete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

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. 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. 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.

2.0.1 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.

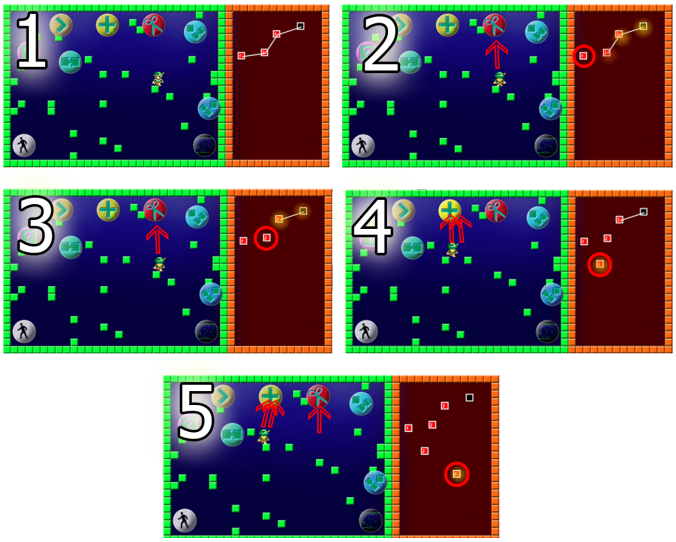

Figure 2: 1) Bob begins with the list [1,2,3]. 2) Bob triggers the pop action, popping the first element from the list. 3) Bob pops the second element from the list. 4) Bob triggers the add action twice, adding 1 and 2, producing 3. Finally, 5) Bob does another pop and another add, producing a 6.

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
```

[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.

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, 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 twofold: 1) 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. 2) 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.

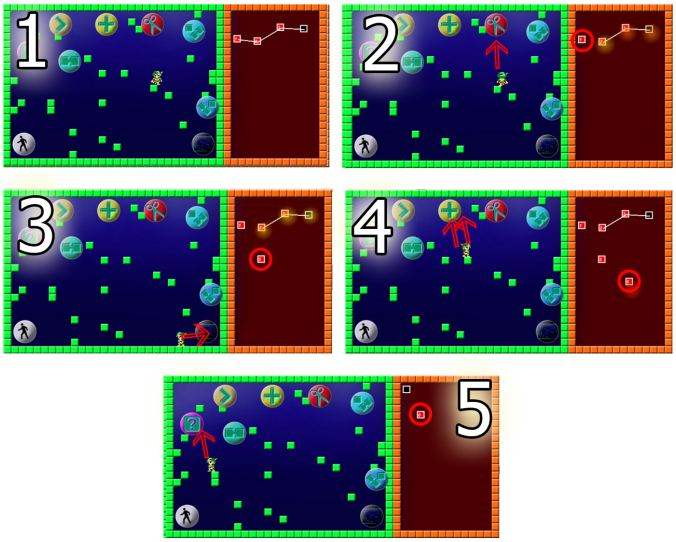

Figure 3: 1) Bob begins with the list [1,2,3]. 2) Bob triggers the pop action, popping the first element from the list. 3) Bob triggers the recursion orb; the oracle returns 5. 4) Bob triggers the add action twice, adding 1 and 5, producing 6, which he returns. Finally, 5) Bob must solve the base case,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

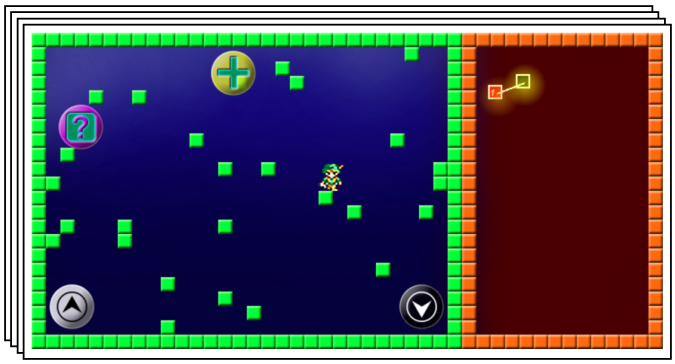

Figure 4: The runtime stack visualized as an actual stack of games during execution. When the avatar triggers the return orb in the lower left, the top-most level will be removed, and the currently selected items in the inventory (the list [1]) will be returned to the next game in the stack.

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.

3. EVALUATION

Because the system is quite unconventional, our main evaluation was a usability study – designed to probe what is usable and what is not about the system.

We recruited 9 novices and trained them for 45 minutes by co-solving various problems – e.g. Adding two numbers together and return the result; popping two numbers off a list, adding them together, and returning the result; etc. We also co-solved three more difficult problems: returning the max element in a list; returning the sum of all the elements in a list; and returning a 1 or 0 value depending on the parity of the number of list items.

We then tested the users on three benchmark problems: reversing the order of the items in a list; returning a list where each number has been increased by five; and returning the last element in a list. We also conducted an exit interview.

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. Furthermore, our exit interview revealed that subjects believed themselves to be playing a video game – not programming.

4. 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. Our usability study shows that it is possible to get non-coders to write (correct) algorithms without knowing it.

5. 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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Prensky, M. *Digital Game-Based Learning*. McGraw-Hill Pub. Co., 2004.