UNIVERSITY OF CALIFORNIA, SAN DIEGO

# Three Paradigms for Mixing Coding and Games: Coding in a Game, Coding as a Game, and Coding for a Game

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Stephen R. Foster

Committee in charge:

2015

## DEDICATION

To all the people.

# EPIGRAPH

*I'm standing*

*right behind you.*

—Anonymous

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

Thanks to Bill and Sorin, my awesome advisors. Thanks to everyone on the committee who made this thesis what it is: Scott Klemmer, Gail Heyman, and Ranjit Jhala. I couldn't have done it without you.

Chapters 3 and 4 use previously published materials. Chapter 5 uses materials submitted for publication.

- Chapter 3 - Stephen R. Foster, Sarah Esper, and William G. Griswold. 2013. 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, 99-108.

- Chapter 3 - Sarah Esper, Stephen R. Foster, and William G. Griswold. 2013. CodeSpells: embodying the metaphor of wizardry for programming. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education* (ITiCSE '13). ACM, New York, NY, USA, 249-254.

- Chapter 4 - Stephen R. Foster, Sorin Lerner, and William G. Griswold. 2015. Seamless Integration of Coding and Gameplay: Writing Code Without Knowing it. In *Proceedings of Foundations of Digital Games* (FDG '15). New York, NY, USA, 3063-3072.

- Chapter 5 - Stephen R. Foster, Sorin Lerner, William G. Griswold. "G(ATS): Games plus automated tutoring systems"

Special thanks to the programmers and artists of ThoughtSTEM who took inspiration from my dinky research prototypes of CodeSpells and LearnToMod, and then went back to the drawingboard to build brand new pieces of software that did so much more than I could have imagined. In fact, these reimaginations go so far beyond what I did as a Ph.D. student that they really shouldn't even bear the same name as the prototypes discussed in this thesis. If this thesis enjoys any sort of wide distribution, I suspect it will be because of the ongoing commercial success of the reimagined CodeSpells and LearnToMod – not anything special about the software I built. So for that reason, thank you to all you talented developers: Jeron Paraiso, Tennyson Holloway, Adrian Lopez-Mobilia, Jason Rosenstock, Caitlin Neff, and Henry Mao. And special thanks to Jeron Paraiso for letting me observe him using his reimagined version of LearnToMod in his classroom at UCSD Extension, and for letting me analyse his class data (discussed in Chapter 5).

## VITA

| | |
|---|---|
| 2009 | B. S. in Computer Science, Southwestern University, Georgetown, TX |
| 2012 | Masters in Computer Science, University of California, San Diego |
| 2015 | Ph. D. in Computer Science, University of California, San Diego |

## PUBLICATIONS

Richard Denman and Stephen Foster. 2009. Using clausal graphs to determine the computational complexity of k-bounded positive one-in-three SAT. *Discrete Appl. Math.* 157, 7 (April 2009), 1655-1659.

Stephen Foster, Walt Potter, Jiang Wu, Bin Hu, and Yu Zhang. 2009. A history sensitive cascade model in diffusion networks. In *Proceedings of the 2009 Spring Simulation Multiconference (SpringSim '09).* Society for Computer Simulation International, San Diego, CA, USA, , Article 5 , 8 pages.

Stephen R. Foster and Ruben Ruiz. 2010. Constructivist Pedagogy Meets Agile Development - A Case Study. In *Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations* (ITNG '10). IEEE Computer Society, Washington, DC, USA, 1092-1096.

Stephen R. Foster, William G. Griswold, and Sorin Lerner. 2012. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *Proceedings of the 34th International Conference on Software Engineering* (ICSE '12). IEEE Press, Piscataway, NJ, USA, 222-232.

Quintin Cutts, Sarah Esper, Marlena Fecho, Stephen R. Foster, and Beth Simon. 2012. The abstraction transition taxonomy: developing desired learning outcomes through the lens of situated cognition. In *Proceedings of the ninth annual international conference on International computing education research* (ICER '12). ACM, New York, NY, USA, 63-70.

Stephen R. Foster, Sarah Esper, and William G. Griswold. 2013. 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, 99-108.

Sarah Esper, Stephen R. Foster, and William G. Griswold. 2013. 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, 305-310.

Sarah Esper, Stephen R. Foster, and William G. Griswold. 2013. CodeSpells: embodying the metaphor of wizardry for programming. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education* (ITiCSE '13). ACM, New York, NY, USA, 249-254.

Sarah Esper, Stephen R. Foster, William G. Griswold, Carlos Herrera, and Wyatt Snyder. 2014. CodeSpells: bridging educational language features with industry-standard languages. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research* (Koli Calling '14). ACM, New York, NY, USA, 05-14.

Sarah Esper, Samantha R. Wood, Stephen R. Foster, Sorin Lerner, and William G. Griswold. 2014. Codespells: how to design quests to teach java concepts. *J. Comput. Sci.* Coll. 29, 4 (April 2014), 114-122.

Sorin Lerner, Stephen R. Foster, and William G. Griswold. 2015. Polymorphic Blocks: Formalism-Inspired UI for Structured Connectors. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (CHI '15). ACM, New York, NY, USA, 3063-3072.

Stephen R. Foster, Sorin Lerner, and William G. Griswold. 2015. Seamless Integration of Coding and Gameplay: Writing Code Without Knowing it. In *Proceedings of Foundations of Digital Games* (FDG '15). New York, NY, USA, 3063-3072.

ABSTRACT OF THE DISSERTATION

# Three Paradigms for Mixing Coding and Games: Coding in a Game, Coding as a Game, and Coding for a Game

by

Stephen R. Foster

Doctor of Philosophy in Computer Science

University of California, San Diego, 2015

Professor William G. Griswold, Co-Chair
Professor Sorin Lerner, Co-Chair

Games for teaching coding have been an educational holy grail since at least the early 1980s [Pat81]. Yet for decades, with games more popular than ever [Pre04, Gee03] and with the need to teach kids coding having been well-recognized [cod14], no blockbuster coding games have arisen (see Chapter 2). Over the years, the research community has made various games for teaching computer science: a survey made by [GB13] shows that most do not teach coding, and of the ones that do teach coding, most are research prototypes (not production-ready) and difficult to even install. In analysing

the list, we found that some were no longer available, of none were blockbusters (see Chapter 2). With decades of unimpressive performance behind us, it is time to take a critical look at the field and ask some key questions: What is a coding game? Why are there no blockbuster coding games? What design guidelines can make it easier to create coding games? How can we categorize past and future work on coding games into a productive taxonomy? What (if anything) makes coding games difficult to produce? What (if anything) makes them difficult to play? What can/should a "good" coding game seek to accomplish? And how can/should we evaluate that?

This thesis begins by articulating a design space consisting of 3 types of coding games. The chapters of this thesis examine those three types in the context of systems we built:

- *Direct embedding – "coding in a game"*. Chapter 3 looks at CodeSpells, a game in which the player plays the part of a wizard and writes magic spells with code. The coding interface is embedded *in* the 3D game. CodeSpells is significant because it introduces a novel set of metaphors (magic, wizards, spells, etc.) that correlate neatly with ideas within the pedagogical domain of coding. As an individual system, it is novel because it is the first fantasy-themed coding game (that we know of); as a more general contribution, it is novel because it validates that fantasy-themed coding games can provide an alternative to the more traditional sci-fi themed coding games.

- *A programming language that is a game – "coding as a game"*. Chapter 4 looks at The Orb Game where a visual programming language is defined as a set of game mechanics, making the process of coding into the process of (seemingly) playing a game. The coding interface is presented *as a* game. The Orb Game is significant because it is the first system that has been proven to allow players to write code without realizing that they are doing so – essentially “tricking” them into thinking they are just playing a game. It is more generally significant because it introduces a novel technique called Programming by Gaming, which can be used to design other Turing-complete programming environments that appear to be games.

- *A modding environment integrated with a game – “coding for a game”*. Chapter 5 looks at a modding environment integrated with the blockbuster game of Minecraft. The coding interface allows the user to code *for* the game. It is significant because it allows players to learn to code in the context of a blockbuster game, resulting in higher recruitment numbers than state of the art coding systems like Scratch. More generally, it proves that loosely-coupled, seamful [CG04] games may be viable alternatives to the more orthodox style of seamlessly integrated games [HA11].

This thesis explores this design space and evaluates the three sub-genres above by looking at real systems we built and evaluated. We also discuss our systems in relation to “adjacent” systems in the same design space. This analysis, in turn, yields design guidelines that we flesh out in Chapter 6, our concluding chapter.

# Chapter 1

# Introduction

## 1.1 The Big Problem

At least theoretically, games that teach coding have untapped potential because 1) children invest a lot of time and self-motivated energy into video games, and 2) the world needs more coders.

In less than a century, computer science has become a domain that cuts across every other STEM discipline – spawning entirely new fields of knowledge: e.g., computational chemistry, bioinformatics, data science, etc. Computing has become integral to academic study, finance, economics, and daily life. The computer is, as Alan Turing dubbed it, the universal machine – a power-tool for automation, communication, and discovery.

Unfortunately, the ubiquity of the computer has increased much more quickly than the public's understanding of it, leaving the majority of the world oblivious to the nature of these machines that shape modern life. The fate and future prosperity of billions of human beings lies in the hands of technology they don't comprehend – or rather, in the hands of the tiny fraction of human beings who do comprehend.

That said, we now look at three ways which in inequity manifests itself: 1) as an American economic trend over time, 2) as a disenfranchisement based on gender and race, and 3) as a deepening of entrenched inequities.

### 1.1.1 American Economy

According to the bureau of labor statistics, by the year 2020, there will be over 1.4 million computing jobs in the American economy [cod14]. There will, however, only be 400,000 workers with a computer science degree from an American university[cod14, cs 15][1]. In other words, American universities are not graduating enough computer science majors to keep up with projected labor demand – creating a deficit of one million unfilled jobs [cod14].

Admittedly, this seems to frame the problem as a purely American problem – or worse, as a problem that big tech companies face. However, upon digging deeper, we begin to uncover the global social justice problem beneath this seemingly America-centric argument. Yes, it is true that big American tech companies will have trouble recruiting talented coders with CS degrees from American universities, which may be bad news for their bottom line. However, tech companies tend have various degrees of freedom to protect their bottom line – e.g., they can fill their labor needs via outsourcing or importing foreign workers.

The real losers are American-born workers who have elected to major in something else and will thus never have a chance to compete for these jobs, and the economy of the countries that are left behind when their STEM-educated citizens leave to work in America (a well-recognized problem known as “braindrain” [GM11] or ”human capital

[1]The source cited here is an NSF report that is fairly encompassing with the term “computer science degree” – including, for example, other computing fields. In examining the report, the only major *not* considered a CS major, but which traditionally trains coders is electrical engineering. EE has less than 50% as many graduates as CS. So if *all* electrical engineering students went into coding jobs, the analysis might change to “600 workers with a computer science degree”.

flight" [HK95]). It is worth mentioning, too, that these 1 million jobs aren't just "jobs". They are careers that have a life-long positive impact on quality of life. Computing jobs tend to involve high pay (an average of $90,000 for software engineers according to Bureaux of Labor Statistics search [bls15] at the time of this writing). And, compared to many other careers, they tend to involve high levels of creativity and low levels of physical danger.

### 1.1.2 Racial and Gender Inequity

Inequity cuts across more than just national borders; it also disenfranchises people across gender and racial lines. The computing fields (both industrial and academic) are dominated by white males [Bad90] – which is both a problem in itself as well as a symptom of other problems: e.g., 1) women in the tech industry are paid disproportionately less than their male counterparts [wom15], 2) the pipeline dropout rates of female and minority students is disproportionately high [JS05], and 3) the computer science AP exam is taken overwhelmingly by white males [EG14]. In other words, an entire pipeline from high school to high paying jobs seems to be systematically leaking women and minorities at multiple stages.

The unfortunate thing about a field being "male-dominated" is that this becomes a self-fulfilling prophesy [PSR99]. Female programmers must routinely be willing to accept being the only woman on their team or in their workplace. This lack of diversity can be detrimental to retention even in the absence of overt sexism or direct harassment [CL06, AR14]. If such factors cause women to leave the field or not to enter into it in the first place, then the diversity problem gets worse or (at best) perpetuates itself.

This lack of diversity spills over into the products produced by the computing industry too. Last year's heated "GamerGate" [HBG14] discussions brought some of these issues to light. The video game industry, for example, routinely produces products that are designed by males for males [Sha15]. These products sexually objectify women, and sometimes even depict violence against them [BBRB12]. Anita Sarkeesian's series of videos called *Tropes vs. Women in Video Games* sought to raise awareness about these issues within the industry and created such a violent backlash of online harassment and direct threats to her life that she was forced to leave her home. The backlash was spearheaded primarily by male gamers who felt that the products they enjoyed were under attack [HBG14, Tri15].

Gamergate was a complex phenomena that is beyond the scope of this thesis, but we do feel that the products produced by the computing industry might be less offensive to women if more of the designers and engineers behind the products were women themselves.

### 1.1.3 Deepening of Entrenched Inequities

We have looked at two forms of inequity above – ones that cut across national borders and ones that cut across lines of gender and/or race. There are no doubt many others ways that inequity manifests itself too. Whatever the manifestation, though, we want to point out that computer science has the unfortunate power to deepen these inequities. We hinted at this in the Gamergate discussion above: products designed overwhelmingly by men run the risk of excluding women. The more general form of this is that innovations tend to serve the innovators themselves.

It is no surprise that when one Googles "productivity software" one can find literally thousands of pieces of software that can improve one's productivity. Why? Because Silicon Valley is a culture obsessed with digital productivity, and people tend to produce innovations based on what they care about. There are 2.2 million farmers in the U.S. and not yet even 1.4 million programmers. Still, Google for "farming software" and you'll find some results – but significantly fewer. Why? Well, the average programmer is better equipped to design software to help the average programmer than to help the average farmer. Designers design best for domains they know best.

Admittedly, we see the same phenomenon in our own work: This thesis is about software that teaches computer science. Why did we choose to teach computer science and not chemistry or economics? Because we are computer scientists, and we went with what we knew and cared about.

This is not necessarily a bad thing. We're not arguing that innovators should *stop* innovating on behalf of themselves and their communities. Rather, our preferred solution is to create more innovators that can innovate on behalf of more communities. As long as the computing field excludes women, there will be fewer innovations that serve women. If the field, over time, excludes millions of Americans, there will be fewer innovations over time that serve Americans. If the field continues to be dominated by white males, then innovations will tend continue to serve white males. If Silicon Valley continues to be the global Mecca for innovative computer scientists, then innovations will tend to serve the average Silicon Valley innovator. And so on.

### 1.1.4 The Hope

Computers have the latent power to equalize the playing field – in particular, through educational software. Software that teaches people about writing software is precisely what this thesis is about. Although we focus on educational games, our motivation is a much loftier goal: To help computing technology become the very bridge that guides people across the educational chasm that it has itself created. Scalable computer science education – delivered through educational software – has the potential to level the playing field by allowing everyone to create software to benefit themselves and their communities, to ensure that technological advancement doesn't leave them behind (or worse, produce externalities that diminish their quality of life over time).

## 1.2 The Video Game Solution

When looking at statistics like those mentioned in the "American Economy Argument" section above, it is tempting to assume that the solution lies in increasing the rate at which colleges graduate coders. But the appropriate question to ask is, why are universities tasked with the job of training coders in the first place? There is nothing intrinsically "college-level" about basic coding. Young people with a few prerequisite skills at reading and arithmetic can (and do) become proficient coders before reaching college – provided that they have access to educational resources. The problem is that these resources are scarce. Coding is rarely taught in high schools and even less often in middle schools. In other words, the problem runs much deeper than university education. With so many young people arriving at university with little to no exposure to coding, only the few who major in it will be viable professional coders when they graduate.

One way of nurturing coding skill among young people is by tapping into the potential of video games. Young people invest a non-trivial amount of time in video games during their college and pre-college years (some estimate it to be as many as 10,000 hours [Pre01]). An ESA study [esa15] notes several demographic qualities that make video games an attractive vehicle for deploying coding instruction: There are 155 million gamers spending about 15 billion dollars per year on games; the population is about 56% male and 44% female; and 26% of them are under the age of 18.

The potential is high. However, there are some concerning issues for educational game designers:

- *People don't consume educational games.* The study above shows that only about 5% of gamers decide to play games because of their educational value.
- *People consume new games frequently.* The ESA study doesn't give us this directly, but we can infer it from the data. The average player is spending about $96 ($15

billion divided by 155 million) on games per year. Assuming the games they are consuming are $50 a piece, that's about 2 games per year. Games tend to cost much less, however. For example, a quick search on steam.steampowered.com revealed that the top-selling games ranged from $15 to $2 – which could indicate between 6 and 48 purchases per year[2]. If we assume that people purchase new games throughout the year (which is at least anecdotally supported by various forum posts we found [pur15a, pur15b]), then people are consuming games every few months at least, and probably abandoning at least some of them in the process.

Why are these things concerning for educational game designers? The first should be obvious: If people don't consume educational games, it makes it less likely that they will adopt the educational games we design. The second problem is more subtle: If people consume new games and abandon old ones frequently, then even if we *do* design an educational game that people like enough to adopt, they may abandon it quickly. This is troubling because we suspect that quick abandonment of an educational game may not leave enough time for the game to accomplish its educational goals. Learning to code takes time (some estimates put the required time as high as 10,000 hours [Pal90]), as we will discuss further in Chapter 2. To incentivise people to spend that much time, we must either strive to design coding games that incentivise many months (or years) of gameplay, and/or we must strive to develop multiple coding games so that users who enjoy them can abandon one and adopt another, and/or we must come up with design guidelines that make it easier to continually produce coding games.

Addressing these problems is critical if we are to ever get a non-trivial portion of the 155 million gamers to spend a non-trivial portion of 10,000 hours playing coding games.

[2]This analysis doesn't even include free games (many of which can be found via Steam and Google searches), which could indicate even more games played per year.

To this end, we propose the following non-exhaustive list of worthwhile endeavors (a larger list is discussed in Chapter 2):

- *Build more coding games*. There simply aren't many coding games [GB13]. The research community has been making such games since the 80s [Pat81], but modern gamers expect newer games that run on the newest hardware [esa15]. We weren't able to run many of the legacy games we found discussed in literature – so the average gamer won't be able to either. To put this in perspective: There are more first-person shooter games on IMDB's top 100 games list [top15] than there are coding games discovered by this fairly rigorous academic survey [GB13]. Simply having more coding games available may increase the odds that users will discover and adopt them.

- *Build coding games that incorporate things that we know gamers already enjoy*. Examples of these are multiplayer mechanics (54% of frequent gamers play in multiplayer mode at least once per week [esa15]) and role-playing mechanics (more than 20% of computer games sold in 2015 were role-playing games [esa15]). Another example would be systems that incorporate already-popular games (there are 100 million registered Minecraft users [mc 15], for example – almost 66% of the total number of gamers given by [esa15]).

- *Derive design guidelines, recipes, or patterns that help designers to create more coding games*. This follows from the above ideas that developing more coding games (with modern graphics and modern hardware) is the main hurdle right now. Any insights that help us develop such games faster or with a lower design burden would serve this goal.

The systems presented in the chapters of this thesis are noteworthy because they do all of the above. They each 1) add to the space of existing coding games, 2) incorporate things we know gamers enjoy (and in ways that prior coding games have not), and 3) imply repeatable patterns for creating similar such games.

The systems we built are also noteworthy because they each lie in a qualitatively different sub-genre within the larger coding game design space (see next section). This brings us to the overarching hypothesis of this thesis:

*Hypothesis: An exploration of three distinct sub-genres within the coding games design space will 1) help produce more coding games, 2) yield novel ways of incorporating into coding games things we know gamers already enjoy, and 3) yield principles that will guide designers in creating future coding games.*

## 1.3 Three Sub-Genres

Note that Chapter 2 contains a more detailed discussion of what a "game" is and what a "coding game" is. But for this discussion, some loose definitions are sufficient: 1) a game is a system that presents the user with a virtual world (which we'll call the "game world", 2) and a coding game is a game that is somehow affected by code that the user constructs. We'll use the term "coding interface" to define anything that the user can use to construct code. And we'll define "code" as any Turing-complete language – visual or textual.

Our classification is one that identifies where the coding interface exists in the system. We can now classify prior coding game systems into a taxonomy with three categories. Furthermore, each category represents a distinct portion of a design space ranging from systems that tightly integrate coding and gameplay, to systems where coding and gameplay are very loosely coupled (see Chapter 2 for a more detailed discussion of seamless vs seamful systems).

There are at least three relationships that the coding interface can have to the rest of the system. Each of these can be summarized by a distinct preposition ("in", "as", and "for"):

- *C[in]G*. Systems that have code *in* a game comprise the majority of coding+game systems. They are games in which coding has been embedded in-game. That is, there is an IDE inside of the game, and the code written there affects the gameplay – e.g. by programming a robot or by creating a magical spell. In Chapter 3, we present CodeSpells, which is a system in this category. *Key litmus test: A system is classified as C[in]G if it 1) contains an IDE which is displayed in-game (either alongside the virtual world, accessible via a modal popup, etc.), and 2) the game world contains more than just the IDE.*

- *C[as]G*. Systems that incorporate coding *as* the game itself are a novel sub-genere of systems, with our own system discussed in Chapter 4 being one of only two we know of. In such a system, the coding *is* the gameplay, and the gameplay *is* the coding. Whereas coding *in* a game, requires the player to mode-switch between code writing and whatever else the game consists of, in a coding *as* a game system, the two activities are (by design) the same, with no mode switch. *Key litmus test: A system is classified as C[as]G if there is an in-game IDE that is 1) not separate from the rest of the game world, but nevertheless 2) still capable of constructing Turing-complete code.*

- *C[for]G*. Systems that provide an environment separate from the game, but capable of changing, "modding", or creating new game are quite common[3], with popular systems like Alice and Scratch fitting into this category. These systems are actually IDEs that are optimized for the creation of games and designed to be approachable for beginners. Also in this category are the various IDEs and tools used to modify

[3]There seems to be a big distinction between creating a new game and modifying an existing one. But, as we discuss in the following sections, this distinction is actually not so clear-cut.

popular games like Skyrim and Minecraft. These systems are not not built for novices; however, we mention them because Chapter 5 presents a novice-friendly IDE we built for modding Minecraft. *Key litmus test: A system is classified as C[in]G if it involves both a game and an IDE, but the IDE is not displayed within the game. It is either separate or separatable from the game.*

We also note that on the spectrum of how tightly integrated the coding and the gameplay are, the above categories can be hierarchically ordered with C[as]G being the most integrated and C[for]G being the least integrated. C[in]G is somewhere in the middle and is generally what people think of when discussing "games for teaching coding".

Systems in the categories of C[in]G and C[for]G are both well-studied (whereas C[as]G is not). We discuss related systems in the following sections.

### 1.3.1 Prior C[in]G Systems

One of the earliest such systems was Karel the Robot, from 1981. In this system, a simple language is used to control a robot that can move on a 2D grid. The API is reminiscent of the Logo language, invented more than a decade before: the "robot" understands instructions like *move*, *turnleft*, and *putbeeper* (similar to putting down the "pen" in Logo). The system inspired numerous variations, many of which are still used today. For example the startup CodeHS teaches JavaScript with an API that manipulates a small "robot" named Karel the Dog.

Perhaps one of the most successful of these "code the bot" games is Lightbot – wherein the player users a simple coding language to control a virtual bot, with the goal of moving it to various targets on a virtual terrain. The game was released in 2008 and has been played 7 million times, according to the Apple app store.

Other systems elect metaphors that are less robot-centric, but still involve the same mechanics. A recently funded startup has released CodeMonkey, in which the player programs a monkey to pick up bananas. The language and API are surprisingly similar to Karel, from the 1980s, and to Logo's programmable dot from 1967, suggesting that little has changed, aside from metaphor, in almost 45 years. Other games use spaceships or tanks. A recently funded Kickstarter game, Codemancer, involves programming a magical tiger that the protagonist rides.

At the end of the day, a game that incorporates coding needs to have *some* game element that is programmable – be it a robot or a monkey, a spaceship or a tiger. So perhaps it is unsurprising that the evolution of dot, to robot, to monkeys and tigers is purely one of metaphor, and not one of mechanics. Dots, robots, monkeys, and tigers can all move forward, turn left, etc. In this regard, they're all "robots" at some level of abstraction. And it's difficult to imagine a game without some kind of "robot" – or programmable entity.

Our own system, CodeSpells, certainly has such entities. Various objects in the game world can be "enchanted", which essentially means that they can be programmed to move, turn left, etc. The novel aspect of CodeSpells is its use of the first-person viewpoint, allowing the player to navigate freely with the typical WASD keys and mouse – i.e. without having to be programmed. As the player moves through the physical environment, he or she encounters various objects that can be enchanted. For example, in one puzzle, the player must jump onto a wooden crate and then enchant it to levitate upwards, so that the player can jump off onto a roof.

We discuss CodeSpells further in Chapter 3, as well as design guidelines for this category of games in Chapter 6.

### 1.3.2 Prior C[as]G Systems

Aside from our own system that we discuss in Chapter 4, we know of only one other system that allows the user to code, but without presenting an IDE that is separate from the rest of the game world. That system is *ToonTalk*, wherein various animated characters act out computations – e.g. birds fly from nest to nest, carrying messages. The player can manipulate how these characters act out computations. At no point does the user see something analagous to text-based code, even in the user-friendly form seen in systems like Alice and Scratch. The act of coding is the act of interacting with various virtual animals in the game world.

Our own system, The Orb Game, described in Chapter 4 casts the code writing process as the movement of an avatar that can walk, jump, and carry items from one "orb" to another. The orbs transform the items that the avatar is carrying. Some orbs implement conditionals (branching in the items being carried), and recursion. While the player moves the avatar from orb to orb, they are programming by demonstration. The program can be replayed on other inputs (items being carried).

In a theoretical sense, C[as]G systems may offer the most seamless experience for the player because he or she doesn't have to mode switch between the IDE and the rest of the game world. However, it is not easy to define an entire IDE based solely on mechanics and objects that generally comprise game worlds. We discuss in Chapter 6 whether this seamlessness has benefits that justify the design burden.

### 1.3.3 Prior C[for]G Systems

Such systems are common and proliferating quickly [DCP06, MRR+10, Mac11]. Scratch [MRR+10] first appeared in 2003 and has since been ported to the web and is used world wide. It is an IDE with a visual programming language and a browsable asset library, allowing students to quickly make 2D games. The software is used in schools and summer camps around the world.

Similarly, Alice [DCP06] provides an IDE for 3D game development, with a visual programming language and an asset library. it was released in 1998 and its Version 3 is still being maintained today. It too is widely used in education at all levels, including the university level.

Other similarly motivated systems are Greenfoot [GM08] and NetLogo [Dic11].

Perhaps most noteworthy – since it is the closest to a Blockbuster game of any educational system we have seen so far – is Project Spark, released in 2014 for Xbox One. It provides an in-game IDE for creating new games. This may tempt one to classify it as C[in]G because the IDE exists in-game. We opted for the C[for]G categorization because it facilitates building stand-alone games that users can share with each other (see the "separate or separatable clause in the C[for]G litmus test). It is the fact that the IDE is not meant to be forever intertwined with gameplay, but rather meant to be abandoned when the user's game is complete (much like published games built with Scratch or Alice) that makes the C[for]G categorization appropriate.

Our own contribution to this sub-genre is an IDE for modding Minecraft – called LearnToMod. This is unique in that it does not try to be a generalized game development tool (like Alice or Scratch), but rather a targeted tool for modifying a single blockbuster game: Minecraft.

## Creating vs Modding

In C[for]G, we have included both systems for creating entirely new games as well as systems for modifying existing games. At first glance, this may seem strange. And indeed, it may seem odd to consider a generalized IDE like Alice or Scratch to be "coding for a game". (One might argue that a term like "coding for games" may be more appropriate.)

We agree that there's a distinction here, but we have chosen to consider the two together for the purposes of this thesis. The justification is that if you watch a student use an IDE like Scratch to do game development, creating a new game very quickly becomes modding an existing game. We have used Scratch to teach coding for many years now, and the following is a typical progression: 1) the coder begins with a blank canvas, 2) the coder adds some thing to the game world (e.g. trees, rocks, an avatar, etc.), 3) the coder introduces a way to interact with the game world (e.g. moving the avatar). At this point, the game has begun to take shape. We have seen students use Scratch to make first-prototypes of games in a matter of minutes. The prototypes may lack features but are certainly recognizable as a game. At this point, further additions can be considered to be modifications to the existing game, and the user is using the system to coding *for a* game. The only difference between modding one's own early prototype and modding a game like Minecraft is that the game of Minecraft has more features.

### 1.3.4 What this thesis is not about

There is much prior research in the field of gamification. Whereas this field is interesting and adjacent to the field of coding games, we will not deal with it directly. Gamification is the borrowing of game-like mechanics to make non-game systems more interesting or motivational. There are a few gamified coding education systems such as CodeHunt and (its predecessor) JavaBat. Although fun, these are more like "fun problem sets", rather than games[4]. Instead, we focus on the use of actual games to give students a context to write code.

[4]Chapter 2 gives a more in-depth definition of "game" – but the main reason these problem sets are not games is the lack of a "game world".

## 1.4 How our systems fit together historically

In hindsight, we now realize that each of our systems exists within a distinct sub-genre of the coding games design space. However, when we were in the trenches building the systems, we were simply trying to "do something different" with each subsequent system. The discovery of the design space in our own work was a happy accident, which we then realized applied to prior art (as shown in the previous section).

We built CodeSpells first, in an attempt to explore a more complex set of metaphors than one finds in the typical "code a robot" type game. In our early user studies with the research version of CodeSpells (which had an embedded Java IDE), we noticed that students much preferred to be running around the game world, rather than coding in Java (perhaps not a surprise).

So we designed The Orb Game to explore the idea that coding and gameplay don't have to be separate. After all, kids can't get distracted by the gameplay experience at the expense of the coding experience if the two experiences are seamlessly combined into one. We succeeded in making a game where code writing and gameplay were one in the same – so much so that users didn't realize they were coding. This was noteworthy from a theoretical standpoint and may inform future systems like it, but we must admit that the game itself was not very fun.

In an effort to make a system that was more fun, we decided not to design another game at all, but rather to borrow a game that was already fun (Minecraft), and to make it educational by designing an IDE for it. The overwhelmingly positive feedback we got from our users gave us confidence that this strategy is a productive one.

All in all, we ended up exploring the full range of possible integrations: coding in a game, coding as a game, and coding for a game. This allows us to draw conclusions about each sub-genre in particular, as well as the design space as a whole. We report on the lessons learned from designing these systems in Chapter 6.

CodeSpells began as a research project, a commercial version began production in 2014 by ThoughtSTEM LLC, following a successful Kickstarter campaign. This version was a ground-up redesign by professional game developers. The new version maintains the metaphor of coding as spellwriting and the name CodeSpells, but is otherwise a completely different game.

Likewise, a commercial version of LearnToMod began production in early 2015 by ThoughtSTEM LLC, a ground-up redesign by professional web developers and Minecraft modders. The commercial version has roughly 10,000 users, provides online coding tutorials, an online community, and a online Blockly-based coding interface.

Although not part of our research investigation, the commercial versions of CodeSpells and LearnToMod can be thought of as a kind of "future work". After all, as we suggested at the beginning of this chapter, it remains to be seen whether or not educational games can compete with traditional games in a competitive market. The long-term experiment is: How will the commercial versions of these educational systems fair? The question can't be answered in the world of academia.

# Chapter 2

# Related Work

## 2.1 Definitions of Games and Coding Games

The definition of "game" is actually deep philosophical territory – first addressed by Wittgenstein in his *Philosophical Investigations* [Wit53]. Since this thesis is not a philosophical text, we'll instead use a more practical definition from video game theorist James Gee: a computer game is a simulation, plus "microcontrol" [Gee07].

To quote Gee in full: *Simulations are regularly used at the cutting edge of science, especially to study complex systems – things like weather systems, atoms, cells, or the rise and fall of civilizations. This raises the question of what differences exist between simulations and video games. There are two key differences: one is that most (but not all) video games have a win state, and the other is that gamers don't just run a simulation, they microcontrol elements inside the simulation (e.g., an avatar in Doom, squads in Full Spectrum Warrior, armies and cities in Rise of Nations, and shapes and movement in Tetris).*

In this thesis, we'll call the simulation aspect of the game the "virtual world". Tetris's virtual world, for example, consists of the blocks and walls, the empty space through which the blocks move, and the simulated gravity that makes the blocks fall. The microcontrol is any way that the player is permitted to interact with the virtual world. We'll usually refer to "microcontrol" as simply "player interaction". And since "microcontrolling the simulation" is a mouthful, we'll usually just say "interacting with the game" or "interacting with the virtual world".

A "coding game" fits nicely into Gee's *game=simulation+microcontrol* definition: in a coding game, a player can microcontrol the simulation (interact with the game) by writing and executing code. It should be noted that Gee's definition allows for games that have multiple forms of microcontrol, and coding games can too. For example, coding games may present the player with an avatar *and* a coding interface – both of which can be used to interact with the virtual world. (CodeSpells and LearnToMod both do this.) But it is also definitionally acceptable for coding games to present *just* code as a means of interaction. (Lightbot is an example of this.)

## 2.2 Educational games

In later sections we'll discuss how coding games (as defined above) can be educational, but this section discusses educational games in general. This section examines academic literature on educational games and comes to the following conclusion: the research community agrees on two things 1) the educational *potential* of games is high, but 2) there is a shortage of design guidelines for creating educational games.

"Motivation" is a typical place to start the discussion on educational games. It is known that when students are motivated, their chances of educational success are higher [ML87, Mal81a]. To clarify, though, the research on motivation distinguishes between two different kinds of motivation – intrinsic and extrinsic – pointing out that "intrinsic motivation" is the kind more likely to lead to positive educational outcomes[ML87].

Intrinsic motivation is when students (or players) are motivated by the act of learning itself, not by some external factor (e.g. gold stars, monetary rewards, threats, etc.). Malone showed that games can engender intrinsic motivation – i.e. that playing them (and consequentially learning the skills required to play them) is its own reward [Mal81a].

Closely related to motivation is the notion of "time-on-task", which has been shown to link to positive educational outcomes [AWP99, Kar84, FW80] – the link being: the more time spent on task, the better. It has been observed (of non-educational games), that games incentivise time-on-task [Gee03, Don07, May07] and have a variety of non-academic educational outcomes (e.g. to teach the skills needed to excel within the game [Pre03, Gee03, Gee05b, KW04]). The trick would seem to be to figure out a way to make the educational outcome be something with direct academic value. Therein lies the challenge (and potential) of designing educational games.

Given that the *potential* of educational games is so well-accepted in literature (see above and [Pre03]), there are surprisingly few empirical studies that investigate that potential with existing systems and/or that actually measure learning outcomes (an observation made by [May07] and [LKLC11]. Indeed, [LKLC11] points out that the focus on "potential" (rather than practical advice) seems to dominate surveys of educational games and game literature [MGBMO$^+$08, Don07]. As [OWB05] points out after a review of thousands of articles on computer games, only 19 were found to demonstrate learning outcomes from educational games. The potential is compelling; the evidence is lacking.

Because of this, educational game designers are in a rough spot: with scant evidence on what educational games work and why, it is difficult to infer good design guidelines when crafting educational games. Thus, we feel (as does [LKLC11]) that the field of educational game design can benefit from evidence-driven design guidelines. And furthermore, because this is true of the superset of educational games, we feel it is equally true of sub-domains like coding games. This is why presenting design guidelines for coding games is one of the focuses of this thesis.

We should note that there is at least one existing design guideline for educational games – and that is "seamless integration". Seamless integration in educational games was first hypothesized to be a desirable quality in 1981 [Mal81a], and there was a rigorous study (in the context of one game) that proved that a more integrated game was more intrinsically motivating than a less integrated form of that game [HA11]. Indeed, in 2011, Linehan named the seamless integration principle as the *one* empirically validated design guideline for educational game designers [LKLC11].

We mention this because our findings with LearnToMod suggest that high levels of motivation can be achieved with a very *non-seamlessly* integrated system. This demonstrates that *seamful* design [CG04] (as opposed to seamless design) is not necessarily a bad thing in the field of educational games. We discuss the implications of this further in our final chapter.

One reading of this thesis work is that we have taken seamless integration (which has been called the one empirically validated design guideline for educational games [LKLC11]), and we have built three systems, each of which adheres to this guideline to a different degree. The Orb Game takes it to the furthest extreme that we were capable of. LearnToMod largely rejects it. And CodeSpells is somewhere in the middle. Our final chapter revisits seamless integration with the benefit of insights gained from these endeavors.

## 2.3 What should a coding game do?

This section makes the argument that 1) learning to code requires many hours of practice, and 2) luckily, encouraging many hours of practice is something that games do well.

Item #2 follows from the time-on-task discussion above. Also, surveys of players [esa15] show that 42% of Americans play games for 3 hours or more per week. Motivating a time investment seems to be one thing that games do well.

On the issue of item #1, it has been suggested that becoming an expert coder may require as many as 10,000 hours of practice [Pal90]. This estimate follows from from Ericsson's work on "deliberate practice" [Eri08, EKTr93], showing that achieving expertise in complex domains requires 10,000 hours. This isn't easy, though. It is difficult even for undergraduates to get enough practice [BM05, Lin92, Pal90]. Palumbo observes that undergraduates computer science students don't get enough deliberate practice when learning to code [Pal90]. It has also been suggested that one of the major problems with undergraduate computer science education is that practice time in labs is not used effectively [BM05]. Others have argued that many fields (not just computer science) have begun to mistakingly prioritize content knowledge acquisition over drilling and practice (whereas athletic and musical domains have resisted this trend) [Lin92]. Surveys of computer science teachers [Dal06] show general educator-frustration with areas of student development that can plausibly be expected to improve with more drilling and practice – i.e. "general maturity" and "problem solving abilities". In part because of lack of practice with large code bases, even recent new hires at top tech companies (i.e. Microsoft) are often unprepared for practical software engineering positions in spite of receiving computer science degrees [BS08].

Note, it is not a digression to examine the shortcomings of undergraduate education here. It is relevant for these reasons: 1) it demonstrates that getting enough practice is a challenge, even when attending an educational institution, 2) it further strengthens the argument from Chapter 1 that students should get practice coding prior to attending college (even if they plan to major in computer science), 3) it validates that getting practice outside of classes is considered necessary by the educational community, and 4) it points out that acquiring content knowledge (e.g. in lectures) is not sufficient for learning to code.

All of this leads us to recommend that the primary mission of coding games should be to get students to *practice coding* (which we define in more detail in the next section). Incentivising practice would appear to be a big problem, even for educational institutions; and prior research on games and motivation/time-on-task would seem to suggest that games may offer a solution to this problem. So it is a logical fit.

## 2.4 How do we evaluate coding games?

As we have mentioned already, coding games face some difficult challenges: 1) only 5% of gamers consider a game's educational value to be a factor in adoption [esa15], and 2) gamers cycle through games frequently (see analysis in Chapter 1). Because of this, coding games (or any educational game for that matter) should be evaluated based on its potential to be adopted and retained by mainstream gamers. We'll call metrics that facilitate adoption or retention *gameplay-related metrics*, discussed in one of the subsections below.

But first, let's begin with the optimistic assumption that the user has adopted the coding game already and will retain it for some time. What then should the coding game seek to accomplish during that time? We'll call metrics related to this issue *learning-related metrics*.

### 2.4.1 Learning-related Metrics

We ended the last major section with the claim that coding games should incentivise coding practice. There are many things that a budding programmer could potentially practice: binary arithmetic, proofs of correctness, historical facts about the history of computing, etc. – but we instead gravitate toward the concepts that are traditionally taught in introductory computer science classes [TG10].

Examples are: loops, sequential execution, variables, conditionals, functions, parameters, user input, etc. There are about 30 such concepts that can be inventoried into 5 categories [TG10]:

- Expressions
- Control structures
- Functions/methods
- Data types
- Object-oriented programming

Together, these are the so-called "CS1 concepts" [CZP14, TG10], all of which pertain to basic coding. These are the kinds of things that Palumbo [Pal90] argues that students receive too little practice with (far short of 10,000 hours).

But given that there are relatively few concepts in the CS1 concepts list, it raises the question of how the ideal student should be spending their 10,000 hours. If it takes far less than 10,000 hours to become exposed to these 30 concepts (and in our teaching experience, this is universally true), what happens after full concept-exposure? Here is where Walker [Wal04] suggests a variety of necessary activities:

- Writing code snippets
- Modifying them
- Making small changes in code and seeing what changes at run time

- Taking existing code and expanding what it does
- Incrementally building up larger and larger programs

We think this is a solid list and have adopted it (along with the CS1 concepts) as inspiration for what a good coding game should do[1]. Namely, a good coding game should: 1) expose students to a range of CS1 concepts, and 2) incentivise continual engagement with those concepts through a variety of code writing, reading, and/or editing experiences.

There are a variety of simple ways to assess to what degree a coding game does this:

- Measure the number of programs users have read, edited, or written while playing the game.
- Look at the corpus of programs that the player has read, edit, or written and identify the number and frequency of CS1 concepts used.

Starting with the above, one could imagine more and more detailed learning outcomes that one might like to see in players – e.g. perhaps things only detectable by administering pre- and post- tests to measure knowledge acquisition or skill gains. In the case of CodeSpells, we did do some of this kind of deeper analysis [EFG$^+$14, EWF$^+$14]. But it is not something we have employed to evaluate our other systems.

One reason for this is that learning is not the only (or even the main) thing that matters in an educational game. As we have already argued, the main challenge is getting gamers to even bother playing a coding game in the first place. If a person doesn't adopt, then the game's potential educational value to that person is moot. Furthermore, a coding game that has relatively little educational value can still be beneficial – e.g. if it causes players to gain awareness and interest in coding games, paving the way for the player to

[1]There are many more concepts in computer science, and many more ways to practice coding. We're not trying to present an exhaustive list here – merely a solid list to get started.

be exposed to other coding games. Because there are so few coding games playable on modern machines [GB13], we think that "the better is the enemy of the good" at this time. Just because something isn't *better*, doesn't mean it can't be *good* for the ecosystem of coding games and for players.

Because of this, we have generally preferred a coarse-grained, binary analysis of learning metrics (rather than a fine-grained one): If the player's corpus of programs contains coding constructs in four out of five CS1 categories, then that's "high concept coverage"; if it contains less, then that's "low concept coverage". As long a game achieves high concept coverage, then we think that's good for the game, good for the player, and good for the ecosystem of coding games as a whole. (Corpus analysis of players in our LearnToMod and CodeSpells studies showed high concept coverage for all players.)

### 2.4.2 Gameplay-related metrics

On the issue of gameplay, we are particularly interested in games that expand the potential audience for coding games. Since we know that only 5% of gamers say educational value is a factor in their video game choices [esa15], coding games should strive to provide value in addition to their educational value. An example might be a coding game that is (somehow) also a first-person shooter. This could plausibly be expected to appeal to a broader audience than 5% of gamers: e.g. 6.4% of computer game sales are shooter games; and 21.7% of console game sales are shooter games [esa15]. Another example: The most common factor in video game adoption is "interesting story or premise" [esa15], so a coding game that (somehow) incorporates an interesting story or premise would be noteworthy too[2].

[2]It should be noted, though, that the actual size of the audience is only something we can empirically measure at great expense (i.e. build a game, make it available, market it widely, and see how many people adopt it). There is still value, though, in estimating the potential audience size by leveraging existing

Note that we said “somehow” in both examples in the previous paragraph. It’s the “somehow” in these kinds of games that we as designers are particularly interested in. How is it that a game might manage to be both a first-person shooter and a coding game? How does the interesting premise incorporate the coding mechanic in the game? Answers to these questions can provide valuable insights to designers when creating more coding games. As we argued in Chapter 1, the more coding games, the better.

LearnToMod, for example, incorporates an already-popular game (Minecraft). There are over 100 million registered users of Minecraft, so that provides an estimated upper-bound for the audience size. This is much higher than the upper bound of the 5% of gamers who adopt games because of their educational value (5% of 155 million is less than 8 million). LearnToMod is able to leverage Minecraft because it loosely couples an IDE with the game – a strategy that could potentially be used to make educational coding environments for other popular games (e.g. Skyrim).

## 2.5 Conclusion

We have already hinted at this, but we want to say it explicitly: *We think the gameplay-related metrics are more important than the learning-related metrics for educational games at this time.* As we have previously identified, the biggest problem facing educational games in general is lack of traction. Perhaps one day, when there are many popular coding games in the ecosystem, and many gamers are playing them, then the learning-related metrics should rise in priority.

That said, we acknowledge that there is a major difference between how traditional education is evaluated and how we are proposing that educational games be evaluated. The reason for this difference is simply that the issue of adoption is less important in

empirical data about what gamers prefer.

traditional education than in educational games. K-12 schooling is mandatory in America. Plus, millions of people are electing to go to college every year [cs 15]. With these sorts of large (sometimes captive) audiences, certain questions become critical: 1) whether people are learning and 2) what they are learning. But with educational games (and perhaps with other domains like MOOCs), where the audience is *not* captive, the issue of incentivising adoption ought to take precedence. With this in mind, we restate from Chapter 1 the goals for designers of coding games:

- Build more coding games[3].
- Build coding games that incorporate things that we know gamers already enjoy.
- Derive design guidelines, recipes, or patterns that help designers to create more coding games.

And for any particular game, we add the following goals, related to learning:

- Validate that games are playable by novice coders, and that players write code while playing.
- Strive to incentivise high concept coverage in the corpus of programs players write.

With these in mind, we can give a quick overview of the contributions of this thesis, which investigates three different coding games:

- CodeSpells. This is a novel system (good: "Build more coding games"). It incorporates mechanics from RPG games, action games, and attempts to have an interesting premise (good: "Incorporate things gamers like"). It maps magic metaphors to coding constructs, e.g. IDE to magic scroll, writing code to crafting spells, running

[3]Note that this is both good for the world, but also for the academic community. More games can pave the way for future research studies that cross-compare the most popular of the existing coding games.

code to casting spells, being a coder to being a wizard, etc. (good: "Reusable design guideline for other coding games"). It also supports multiplayer (good: "Incorporate things gamers like"), and the multiplayer "mode" is actually a separate, standalone game (good: "Build more coding games"). And we demonstrate how competitive multiplayer communities can be incentivised to write increasingly complex code in the context of coding games (good: "High concept coverage" and "Reusable design insights") We validated that the single-player version incentivises player corpuses with high concept coverage [EWF$^+$14, EFG$^+$14], and so does the multiplayer version [FEG13].

- LearnToMod. This is a novel system (good: "Build more coding games"). It takes a blockbuster game and transforms it into a coding game (good: "Incorporate things gamers like"). It incentivises students to write code in their free time (good: "Playable by novices", bonus: "Intrinsically motivating"), and the code uses many CS1 constructs (good: "High concept coverage"). Furthermore, it introduces the idea of bolting an IDE onto an game, a technique that can potentially transform other popular games into coding games (good: "Reusable design insight").

- The Orb Game. This is a novel system (good: "Build more coding games"). It is a platformer game like Mario (good: "Incorporate things gamers like"). But the concept coverage it was capable of incentivising was relatively low (bad) and the game wasn't very easy to play (bad). On the other hand, The Orb Game is a more seamlessly integrated system than any other coding game we know of, allowing us to add to the ongoing seamless integration conversation (good: "Reusable design insights").

Holistically speaking, the three systems together represent an investigation of seamless integration (what Linehan calls the one empirically validated design guideline for educational games [LKLC11]). As such, this thesis contributes to the larger conversation on the design of educational games. We flesh out the design space around the principle of seamless integration, as it pertains to coding games. We demonstrate what it is like, from the perspective of a designer, to adhere (or not) to the seamless integration principle while building full-fledged coding gmaes. We show that there may be justifiable reasons to abandon seamless integration (e.g. for the sake of lowering the design burden). We show that abandoning it can still yield systems that incentivise high levels of intrinsic motivation – showing that even if seamless design is better, seamful design is not necessarily bad. These are novel design ideas in a field that is in need of more design guidance [LKLC11].

# Chapter 3

# CodeSpells: Coding in a Game

CodeSpells has generated a variety of publications [FEG13, EFG13a, EFG$^+$14, EWF$^+$14]. This chapter summarizes that work in three parts, the first part discusses related work, the second part discusses singleplayer CodeSpells, and the third part discusses multiplayer CodeSpells.

Some high-level contributions of this chapter are:

- Description of how magic metaphors can permeate a coding game – influencing the design of subsystems such as the API, the IDE, and in-game learning resources. Our design decisions are generalizable beyond CodeSpells and can be applied to other coding games that employ magic metaphors – a worthwhile endeavor, given the popularity of role-playing adventure games [esa15].

- Results of surveys that characterize certain chess and StarCraft II communities, suggesting that communities like these engage in beneficial pedagogical activities and have a positive “mindset” [Dwe07]. This advances Gee’s well-known claim that various non-educational games are powerful tools for learning [Gee03], proving that such learning benefits may also extend to communities surrounding the game.

- Results of a lab study in which we created a small competitive community around multiplayer CodeSpells and observed the community's investigation of the competitive CodeSpells strategy space. We characterize that strategy space and the pedagogical benefits to the individuals who investigate such a space. Various insights are generalizable to the design of other competitive multiplayer coding games: 1) simple game mechanics can yield complex, emergent multiplayer gameplay, 2) the gameplay need not be directly educational; rather the exploration of the strategy space around the game can be educational, and 3) designging a interesting multiplayer experience can actually have a lower design burden than designing a singleplayer experience.

## 3.1 Related Work

### 3.1.1 The Language Debate: Accessibility vs Authenticity

Languages such as Pascal and BASIC were early efforts to make syntax more appealing to novices [KP05a]. However, due to a growing preference for industry standard languages [dRWT03], these educational languages have fallen out of vogue in most American universities. Rather, much of the educational community currently finds itself embroiled in an Optimal Novice Language Debate over which industry standard language helps strike the the right balance between palatability and authenticity [dRWT03, Bla11, Jab07, MdR06, Sol78, Hon98]

In spite of the trend toward aligning education with industry standards (i.e. making education more authentic), there still exists an opposite trend (in a similar vein as Pascal and BASIC) toward eliminating the written program entirely, replacing text-based code with virtual blocks that fit together. Examples of such tools are Alice [DCP06], Scratch [MRR$^+$10], and Lego Mindstorms [leg12]. Systems like Toque [TSD$^+$10] and Tern

[HSCJ09] represent even greater departures from text-based coding, allowing programs to be constructed by acting out cooking actions (in front of a Kinect) and by by assembling physical blocks. In all of these systems, though, the composition of the program is non-textual, causing it to become less authentic (in the sense that it is not industry-standard).

There are results that show that languages like Scratch allow students to better understand concepts, such as conditionals [Lew10]. However, proponents of teaching industry standard languages argue that the goal of coding instruction is to pave the way for students entering industry, and so the language of instruction should be inspired by industry [dRWT03]. We think there are good points on both sides of the debate. Our decision to employ Java as the in-game language of CodeSpells was driven not by a belief that Java is better for novices, but rather by a belief that if CodeSpells could make a language as syntactically verbose as Java palatable for novices, then the same techniques could plausibly be expected to make other, more accessible languages palatable too.

### 3.1.2 Learning Theory

In the part of this chapter that investigates multiplayer, the investigation focuses on the learning environments that arise from community interactions centered around 1-on-1 competitive games. From this perspective, the intellectual lineage of our approach can be said to begin with Vygotskian learning theory [VC78], which places great import upon the social contexts of learning – both for the purposes of understanding learning and for designing new learning environments.

There also several other ideas from the learning sciences that have guided our analysis of competitive learning environments:

### Metacognition

As this chapter unfolds, we will take a particular interest in metacognition – the activity of reflecting on one's own learning process. Metacognition has been widely recognized by the research community as a critical factor in optimizing both the depth of learning [WF98, SMAoA83], as well as the ability to transfer learned knowledge to new contexts [SBS84, Fis98]. Indeed, one of the key arguments in the Research Council report *How People Learn* was that learning, teaching, and the construction of learning environments can all be improved by taking into account the importance of metacognition and explicitly promoting it [BoDitSoLoLRP00].

### Self-theory

Related to metacognition is the notion of self-theory – the beliefs a learner possesses regarding his or her own ability to learn. As famously claimed by Carol Dweck, there exist at least two distinct outlooks or "mindsets" toward learning – a growth-mindset and a fixed-mindset [Dwe07]. A growth-mindset is one in which an individual believes she can improve, which correlates with high levels of resilience to failure and ultimately with high levels of success. A fixed-mindset, by contrast, is characterized by the belief that one's intelligence or skill is a fixed quantity, which tends to have a detrimental effect on one's acquisition of skills. The importance of these mindsets has been studied quite extensively in academic disciplines such as computer science [SHM$^+$08, MT08].

### Feedback

It has been observed that effective metacognition can be enabled by effective feedback [SBS84, Fis98]. High quality feedback that is timely and formative (rather than summative) can expose the innerworkings of a learner's process to that learner, which can in turn facilitate self-reflection [BoDitSoLoLRP00]. It has also been observed that video games make effective learning tools precisely because they give constant and immediate feedback to the player [GS10]. Such feedback allows for effective metacognition through accurate self-modeling of one's own strengths and weaknesses [Sad10].

Dweck has argued that certain types of feedback can be beneficial in relation to positive self-theory as well. Feedback that reinforces a fixed-mindset (e.g. "You're so smart!") should be supplanted by feedback that reinforces a growth-mindset (e.g. "You've worked really hard!") [1].

Because of the importance of feedback in both metacognition and self-theory and because good video games have such effective feedback mechanisms, it becomes evident why even so-called "non-educational" games have been identified as effective learning tools [Gee05a]. We will see that 1-on-1 competition offers unique benefits in terms of feedback. In particular, 1-on-1 competition is a feedback mechanism that scales particularly well in the face of high diversity and sustainability.

### 3.1.3 CodeSpells

From the beginning, *CodeSpells* was intended to be a concrete instantiation of the prevalent theory that educational games should interweave educational subject matter as seamlessly as possible into the fabric of the game [IFH10]. The point of taking this approach is, of course, to leverage the addictive qualities of a well-designed game to inspire students to spend as much time and effort on educational activities as the average gamer spends on games. However, we also wanted to explicitly design for authenticity – e.g. to use Java[2].

*CodeSpells* includes various tropes commonly found in other RPGs:

[1] We're just borrowing some classic Dweckian examples here. It should be noted that neither of these feedbacks is necessarily good for metacognition. But there's no reason why the ideas of metacognition and self-theory can't potentially be combined to derive even better feedback.

[2] As we mentioned earlier, this was primarily to strengthen the result: If we could make Java palatable, we believed we could also do so with less syntactically verbose languages.

*Quests*. Many role-playing games intentionally lack "levels" that must be completed in sequence. Instead, the basic building block of an RPG is the "quest" – a goal or series of goals with a reward for completion that roughly matches the time invested by the player. Quests can often be completed in a variety of orders, although the completion of some quests may unlock the privilege of embarking on previously unavailable quests. Thus, instead of a linear sequence of levels, RPGs more commonly exhibit a directed, acyclic graph of quests

*Non-Player Characters (NPCs)*. Whether the RPG is a multiplayer game (e.g. *World of Warcraft*) or a single player game (e.g. *Skyrim*), the game is usually against a the backdrop of a fantasy world populated by NPCs with whom the player can banter, barter, and even battle. The NPCs are the primary the origin points for quests: promising a rewards when certain deeds are done.

*Magic*. Most fantasy RPGs feature a suite of magic spells that the player can employ – either innately or by way of magic scrolls which may be used even by non-magical characters. These spells exhibit many similarities across games. Hurling balls of fire at one's enemies, for example, is common practice. This is the only trope for which *CodeSpells* makes an explicit departure from the fantasy RPG genre: In addition to giving the player a suite of predefined spells; we also give the user in-game access to an API that allows the user to craft her own magic spells. It is precisely this relatively simple adjustment to the popular RPG genre that fuels the plot, gameplay, and educational qualities of *CodeSpells*.

Magic spells in *CodeSpells* are representated as Java programs. In the world of the game, they are more than that, though. The spells are integrated into the quests, plot, physical environment, and storyline of the game. Spells are crafted through a streamlined in-game IDE which also facilitates an authentic dialog with the Java compiler and with our *CodeSpells* badge engine. Users are given an extensive spellbook whose layout and contents explicitly scaffolds the kind of internet resources that professionals find in online JavaDocs, open source projects, and Stack Overflow.

### 3.1.4 Spells as Java Code

Because syntax is such a critical component in the interface between programmer and program, we now give an overview of the *CodeSpells* API and the syntactic artifacts (spells) that can be crafted with the API. Here is an example spell – indeed, the first spell in the spellbook given to the player when the game begins:

```
import codespells.*;

public class Flame extends Spell{
  public void cast(){
    Enchanted thing = getTarget();

    thing.onFire(true);
  }
}
```

Spells are structured as Java classes with a *cast()* method. Note that we do not hide the "import" statement nor the class and method definitions (though this would be trivial to do). In the interest of authenticity, we elected to use only standard techniques from software engineering, combined with game mechanics to improve the Java interface.

For example, those familiar with Java will notice that we make efforts to hide executional complexities inside of the game mechanics. For example, there is never a need for a player to see "public static void main". Instead, simple spells are executed by dragging a scroll from the player's in-game inventory on to game objects. In the background, *CodeSpells* asks the javac to compile the spell class and runs a Java program that constructs the spell object, executing the *cast()* method within the Java virtual machine and communicating state-changes to the *CodeSpells* game via a language-agnostic network socket protocol.

Additionally, we have made every effort to make the API both simple and intuitive for novices, while retaining sufficient power. During our design phase, we defined "sufficient power" as the power to replicate and extend spells from popular games. Our API design process involved working extensively with an intermediate programmer (a first-year college student) over the course of three months. He was asked to write spells to perform various tasks, such as to recreate spells from popular games: setting things on fire (*Skyrim*, *World of Warcraft*), changing the weather (*Skyrim*), telekinesis (*Ultima Underworld*, *Knights of the Old Republic*), teleportation (*Portal*, *Daggerfall*), levitation (*LEGO Harry Potter*, *Ultima Underworld*), flight (*Daggerfall*), area of effect fire (*Skyrim*, *World of Warcraft*), fire traps (*Skyrim*), building a bridge of rocks over a river (unique to *CodeSpells*).

Our self-imposed requirement was that no Java file for implementing one of these common spells should be longer than 10 lines. Note: This was somewhat arbitrary and based on intuition, having designed (and redesigned) many APIs in industry settings. A more in-depth theoretical treatment on the design of APIs can be found in [Hen07, SGSZ11].

With these constraints in mind, we observed our intern's progress, taking extensive notes especially regarding occasions where he struggled with aspects of the Java language or with translating his ideas into code. On these occasions, we strove to derive software engineering solutions to help others avoid similar struggles. We will list some of these struggles and their solutions here momentarily. But first we outline our high-level approach in more detail in the next section.

## Our Heuristic for Accessibility

We should take a moment to return to the accessibility vs authenticity discussion we introduced earlier. Although Java is an authentic language, we do (as shown above) take various measures to make it accessible in the context of the game. Our heuristic for what measures to take can be boiled down to the following: *We only used industry-standard techniques for increasing the accessibility of Java in the context of CodeSpells*. Writing APIs for specific domains a standard technique (see the Java API for interfacing with OpenGL, for example). Removing the main method in favor of a more domain-specific code entry point can be seen in other Java systems (see the Android platform).

As with our decision to use Java, we used the above heuristic to ensure that the authenticity of the Java experience remained high. There are potentially good learning theoretical reasons to do so (e.g. better transfer of knowledge to new Java contexts [BoDitSoLoLRP00]), we chose to do this largely to strengthen the result: If CodeSpells can make an authentic Java experience fun and accessible, then CodeSpells can plausibly be expected to also do so with more accessible languages (like Scratch) too.

The reusable insight here, though, is that if you happen to be in the camp that prefers teaching with authentic, industry standard languages (as the majority of educators are [dRWT03]), then the heuristics we used to incorporate Java into CodeSpells can be reused in other systems that are also striving for high levels of authenticity.

## Entity Access/Reference/Traceability

Obtaining a Java reference to particular objects in the 3D world sounds trivial but is an interesting interface problem. We handled the trivial case by folding some complexity into the game mechanics. When the user drags a spell onto a particular target object, that object may be referred to by the *getTarget()* function.

```
Enchanted e = getTarget();
```

However, during the course of working with our intermediate programmer, it quickly became necessary to refer to more than simply one game object. We added the ability to reference certain special objects by name. For example, the player may obtain a reference to their own in-game character as follows:

```
Enchanted e = getByName("Me");
```

However, this raises a new challenge: How do objects get their names? We leveraged a pretty standard technique here – which is to give predefined names to certain permanent fixtures in our environment – i.e. “Me” and “Rain”. The Unity game engine, for example, uses the same kind of technique for getting references to game objects.

The following spell can be used to make the local rain cloud follow the player around:

```
import codespells.*;

public class SummonRain extends Spell{
  public void cast(){
    Enchanted rain  = getByName("Rain");
    Enchanted myself  = getByName("Me");

    while(true){
      Direction toward_me =
         Direction.between(rain,myself);
      rain.move(toward_me, 5);
    }
  }
}
```

To facilitate more complex name-based code references, players may also give names to objects that they can pick up. So if a player wished to refer to a particular rock in a spell, she could give the rock a name and use the above syntactic construct to obtain a reference to the rock.

These methods make the API relatively flexible, but it turned out not to be quite sufficient. When building a bridge out of 20 rocks, for example, it turned out to be tedious to give names to every rock and to declare ten variables of type *Enchanted*. We handled this by adding API support for defining variably sized lists of objects.

**Abstract lists of objects**

Our first attempt at allowing the user to specify a variably sized list of game objects was to use a "reference object" – for example:

```
import codespells.*;

public class FireAOE extends Spell{
  public void cast(){
    Enchanted reference
      = getTarget();
    EnchantedList list
      = reference.getWithinRadius(10);

    list.onFire(true);
  }
}
```

This spell constructs a list where each *Enchanted* element in the list is within a distance of 10 from the user-targeted object and catches all of them on fire. This allows the API to support a classic spell-type in fantasy RPGs: the Area of Effect (AoE) spell – where one's magic takes effect on anything within a certain area.

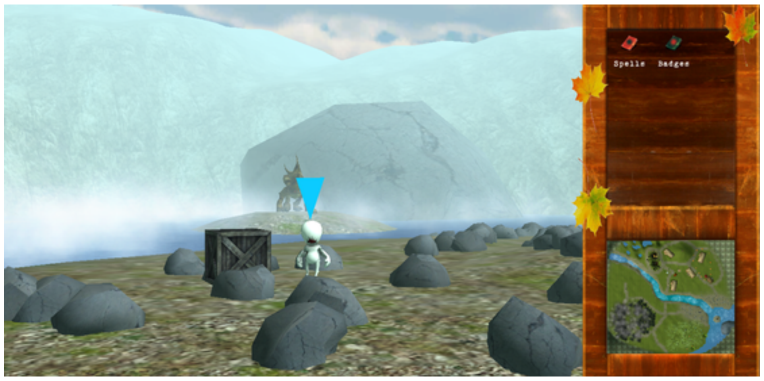


**Figure 3.1**: It is difficult to visually assess how many rocks are within a particular radius.

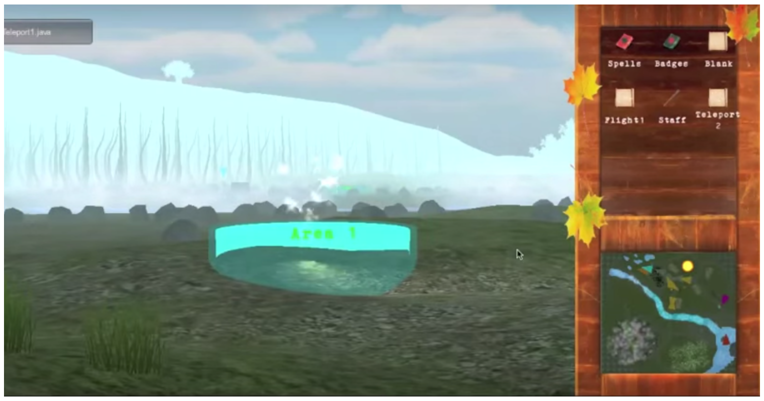


**Figure 3.2**: With a visual area in game, it is easier to see how many rocks are within the radius (zero in this case).

However for constructive tasks, such as building a bridge of rocks the above was still not sufficient. The *getWithinRadius()* method resulted in numerous errors that were hard for our intern to debug. It was all too easy to accidentally obtain too many rocks, or too few. The cognitive root of the problem is that a radius of X is not necessarily easy to visualize before casting one's spell (see figure 3.1). To ease the burden on the programmer, we again resorted to folding some of the complexity in the game mechanics.

We elected to give players a magic staff which may be used to designate static areas of effect (see figure 3.2). These areas may be resized, renamed, and may be referred to from within one's spells using either the *getTarget()* or the *getByName()* functions we have already discussed. The following is a spell which may be used to lay a permanent trap for one's enemies (a spell commonly referred to as a "ward" or a "rune" and seen in many of the *Elder Scrolls* games, such as *Skyrim*):

```
import codespells.*;

public class FireTrap extends Spell{
  public void cast(){
    Enchanted area
      = getByName("Fire Trap Area");

    while(true) {
      EnchantedList list
        = area.getWithin();

       list.onFire(true);
    }
  }
}
```

These areas may also be used, for example, to create a portal from one area to another – similar to the game mechanic that undergirds the game *Portal*.

Allowing the player to designate areas that could be seen in-game was sufficient to allow our intern to craft rather difficult spells – such as building a bridge out of rocks – without producing bloated code or without spending an overly long time debugging.

## Vector math

Another challenge was to allow novices to manipulate objects in 3D space without requiring knowledge of vector mathematics, linear algebra, trigonometry, and (preferably) not even geometry. Thus, our API, as a syntax-based artifact had to provide intuitive ways of performing spatial manipulations without exposing mathematical syntax. Instead, our API allows the user to define directions relative to him or herself. This allows us to leverage the state of embodiment in video games in order to simplify the API in a way that would not be possible in an environment that did not simulate the user's embodiment. In CodeSpells, there are six absolute directions – up, down, north, south, east, and west – and four relative directions – backward, forward, left, and right. This is *somewhat* similar to how systems like Alice allow users to control in-game objects, except that Alice does not embody the user in the game, so the semantics of "left", "right", "up", "down", etc. are relative to whichever object is moving. In CodeSpells, they are always relative to the player's avatar. We're not making a claim about which is better – just identifying the difference.

The following spell uses the "forward" direction to allow the player to fly when standing on top of another object (e.g. a magic carpet). Because "forward" is dependent on the directly the player is currently facing, the behavior of the spell is affected at runtime by the player's movements, flying in whichever direction the player's embodied avatar is currently facing.

```
import codespells.*;

public class Flight extends Spell
{
  public void cast()
  {
    Enchanted target = getTarget();

    while(true){
       target
       .move(Direction.forward(), 0.2);
    }
  }
}
```

It is worth mentioning that, because these spells are true Java classes, each spell becomes a building block which may be constructed and used in more complex spells.

### 3.1.5 Program as Spells

Programs in *CodeSpells* are, of course, spells – and this metaphor is a critical part of the interface. Magic is the ability to break physical laws within one's vicinity, often for "fun and profit". The celebration of magic is the celebration of a collection of fantasies that have motivated the sales of video games for more than 20 years (and books for centuries).

However, in addition to being a motivational tool, the magic metaphor has a number of beneficial qualities in the context of computer science education:

*Magic is portrayed as an academic subject in literature.* Spellbooks and Grimoires are simply textbooks for magic spells. The stereotypical powerful wizard owns and has poured over hundreds if not thousands of ancient tomes, practiced for hours, and often performed original research. Academic institutions, such as Hogwarts of *Harry Potter* and the Mages' Guild of *The Elder Scrolls* are readily accepted motifs in books and games. For the learner of computer science, the magic metaphor and its typical portrayal in Western literature implies many of the qualities that correlate with academic success: hard work, the acquisition of domain knowledge, trial and error, and hours of practice.

*Magic is supposed to be mysterious.* Magic (or the "mystic arts") are not meant to be fully intelligible, at least not for the uninitiated. Likewise, many aspects of programming and computation are mysterious to beginners. The magic metaphor implicitly conveys the normalcy of confusion and short-circuits the novice misconception that if she doesn't immediately "get it" then it's "not her thing". One would hardly have expected Harry Potter to immediately know everything required to vanquish Voldemort.

### 3.1.6 Embodiment

Another interesting concern is the runtime interaction between player and spell. After a spell is cast, the effect on the player's world is a temporal phenomenon. Take, for example, the Flight spell seen above. This allows the player to fly when standing atop an enchanted object. The inputs and outputs of a *CodeSpells* program are state changes in the 3D game world.

The input is the direction that the player is facing at any given time during the execution of the spell. The output is that the enchanted object moves in the direction that the player is facing. A program that operates upon a 3D world is already a significant step toward making programming meaningful and tangible for learners. However, *CodeSpells* programs are qualitatively different from the 3D outputs of, say, Alice programs. Whereas Alice has a camera, CodeSpells has an avatar. The difference is that the avatar can interact with the 3D world. The avatar embodies the player within the world that the program operates upon. Thus the programs take on a much more personal significance. By jumping on top of an object and casting the above spell, the player can live out the fantasy of flight – navigating in midair by changing the direction in which she is looking. The player's embodiment plus the ability to interact with spells at runtime is what make this possible.

#### Quests and requirements

In software engineering terminology, the "requirements" of a user's program are set forth by quests initiated by NPCs. The user is asked to meet these requirements however she sees fit (and indeed, there are numerous solutions to many of the quests).

We outline a few of these quests here. The can may be geared toward absolute Java beginners by providing the solution spells in the spellbook, requiring the user simply to locate and execute the correct spell. However, the difficulty of the puzzles can be increased by not including certain spells in the spellbook, or by providing incomplete or broken spells.

*Firestarter*

In the Firestarter quest, an NPC expresses to the player that he is jealous of his brother who owns the village's largest firepit – which used for smelting the village's raw materials. He asks the young wizard to create a bonfire in a nearby pit. One solution is to use Flame spell in the spellbook (if it is made available to the player) and to cast it on a few of the flammable objects in the pit.

*Crossing the River*

The player begins the game on one side of a wide river that divides the playable area into two parts. Entering the river awakens a river monster, who will hunt the player down and eat the player (initiating a "respawn" at the beginning of the level). One quest involves using magic to get to the other side of the river. The puzzle can be solved in various ways – for example, by using the Flight or Teleportation spells mentioned previously, or by building a bridge, or by creating a moving platform. Each of these solutions, in fact, arose during our user study.

*Getting High*

The player soon finds out that she cannot cast spells on herself. However, she is able to levitate objects. By jumping atop an object prior to levitating it, though, she can move herself upwards too. This is the solution to a quest that requires the user to speak with a watchman standing atop the chimney of the tallest building in the village. Although the player's spellbook contains a Levitation spell, it must be modified in such a way that it reaches the correct height.

*Collecting Bread*

One NPC has botched a magic spell and caused several loaves of his bread to float into the air. He asks the player to collect the bread on his behalf. This quest's solution is an extension of the previous quest, except that the levitating object on which the player is standing must be moved laterally as well. The Flight spell mentioned previously provides one solution to the puzzle.

These are just a few examples – all of which are built upon *CodeSpells*' extensible quest-building platform. It is entirely possible for educators to create new quests to illustrate specific concepts. Indeed, we are currently utilizing *CodeSpells* in an academic course where this is the primary mode of instruction.

## 3.1.7 Automated Feedback in *CodeSpells*

#### Negative Feedback

For the time being, the compiler and runtime feedback in *CodeSpells* is similar in nature to the feedback given to professional programmers. That is, we report compiler errors to the user in exactly the form in which they are produced by Sun's Java compiler. Runtime errors are similarly unsuppressed. For example, if a user tries to refer to an object in the world that doesn't exist (e.g.

```
Enchanted e = getByName("MumboJumbo");
```

), the spell will fail with whatever message is produced by the Java virtual machine. The cryptic nature of these messages did not appear to concern the subjects of our studies on the single-player version of CodeSpells [EFG$^+$14, EWF$^+$14]. Rather than making the error messages less authentic, we chose to augment the automated feedback with positive output as well.

#### Positive Feedback

CodeSpells does, however, have integrated support for awarding "badges" – small tokens of achievement for overcoming various challenges. Badges are given for completing quests, but also for programming-related activities: editing code for the first time; writing your first while-loop; fixing your first bug. The IDE and the Badge-engine are integrated such that one can implement badges for certain errors – e.g. a program that fails due to a line like *getByName("MumboJumbo")* can award a "Nice Try" badge. The goal is to encourage exploration by making certain "failures" into Easter eggs. We felt that the addition of even a small amount of positive feedback to the programming dialog experience would be an effective way to leverage standard techniques of game design to help make a potentially disheartening experience into positive one, without removing the authentic error messages.

### 3.1.8 The IDE in *CodeSpells*

The *CodeSpells* IDE is relatively streamlined and errs on the side of fewer features. Naturally, it allows for spells to be opened and edited. During the editing process, it performs real-time compilation and marks errors in the user's code much like the Eclipse IDE or any standard Java IDE. There is no need for the player to explicitly trigger the compilation process. Additionally, it provides minimal syntax highlighting support. Finally, as can be seen in the figure, the artistic design of the IDE is intended to reinforce the magic metaphor: players enscribe their spells onto a scroll. Scrolls can later be used in a way that conforms with relatively typical RPG game mechanics: Spell scrolls icons can be dragged onto the player's intended target in the 3D world, which triggers the execution of the Java program.

It should be noted that although our in-game IDE does not exhibit many features, it does have the ability to interface with the Eclipse IDE running as a background process, allowing for future augmentation. For example, our summer intern experimented with using a background Eclipse IDE to enable method name completion functionality within the *CodeSpells* IDE.

### 3.1.9 Community Support in *CodeSpells*

CodeSpells is brand new, so naturally there are no organic community resources. However, we do simulate this external support with what is arguably one of the most important aspects of *CodeSpells*: the spellbook.

The spellbook is a compilation of spells designed by us and includes complete source code for each spell as well as extensive documentation about each line of code. It is, in essence, an in-game textbook with "lessons" that are driven entirely by useful spells which serve as examples. Each spell may be copied from the book with the click of a button and edited by the player in the in-game IDE.

Although these spells are crafted by the game designers, and not provided by an internet community, the educational significance is still non-trivial. Expert programmers routinely read code online, browse documentation, and employ cut-paste-modify techniques both for the sake of learning and in order to incorporate snippets of code into their own codebases [BDWK10]. It is precisely this kind of beneficial interaction with internet resources that the spellbook is intended to encourage. Indeed, the player may copy an example spell from the spellbook to the IDE with a single button click. Through the spellbook, players are given the authentic experience of interacting with someone else's code in a variety of ways: spellbook spells may be executed as-is (used as blackboxes), or may be lightly tweaked or heavily modified (used as artifacts). In both cases, the user is engaging in an authentic practice with Java programs.

Furthermore, the spellbook is web-enabled, facilitating the dynamic addition of new spells that may be created by the community in the future, as well as the direct sharing of spells between players. Spells may also be "unlocked" as rewards for completing quests.

## 3.2 Multiplayer

The singleplayer CodeSpells experience has been evaluated and published upon numerous times [EFG$^+$14, EWF$^+$14, EFG13b]. The major findings are that CodeSpells 1) engages students in spite of the syntactic difficulties of Java, 2) learn CS1 computing concepts, and 3) has enough single-player content (quests) to allow for between 4 to 10 hours of gameplay (for students 9 to 10 years old). We also show that the design of singleplayer "quests" is a non-trivial design problem [EWF$^+$14].

What we present here is the research on the emergent properties of CodeSpells multiplayer. Perhaps the biggest quantifiable takeaway from this section is that multiplayer can yield more replay value for gamers, and at a lower design cost for designers. The main reason is that it's non-trivial for a designer to put engaging content into a game [EWF$^+$14]. But in a competitive multiplayer setting, the engaging content arises from the fact that there are people to play against.

### 3.2.1 Two Design Principles

A classic and previously criticized [Bru99] exemplar of educational games is the game of *Math Blaster*, which sought to teach the academic subject of arithmetic through repetitive drilling. The game serves up arithmetic problems, seeking to make the process fun by conspicuously interleaving these drills with much more engaging (non-academic) gameplay. Such games have been criticised as "chocolate-covered broccoli" [LKLC11, Bru99].

In recognition that such educational games may not be living up to the genre's full potential, designers have more recently advanced the aesthetic principle of "seamlessly interweaving" academic subject matter with gameplay [Mal81b, IFH10]. One exemplar of an "interwoven" educational game is, *Zombie Division* [HA11, HAB05], which seeks to teach division by placing players into a virtual environment with zombies (who have numbers on their chests) and weapons (which also have numbers). Students must employ the weapon whose number divides into the zombie's number in order to defeat the zombie. Similarly, the recent game *Refraction* [SAMP12] seeks to teach fractions by giving players problems that can be solved by splitting (and thus dividing) beams of light in order to power spaceships. In both of these games, the academic content is tightly woven with gameplay, making the games quite engaging.

While the aesthetic trend away from "chocolate broccoli" style games does make games more enjoyable, we take the position that there are at least two critical concerns which have not been adequately addressed by the "seamless interweaving" trend, and which are economically difficult to address in single-player games at all:

*Diversity*. It has been observed that "chocolate broccoli" games are not fun (compared to other games) [Bru99], but they exhibit a narrow pedagogical scope. Arithmetic, for example, is only a small part of what mathematics education is all about. For that matter, solving out-of-context arithmetic problems is only a small part of what *arithmetic* education is all about. Interestingly, second-wave educational games such as *Zombie Division* and *Refraction* – in seeking to "seamlessly interweave" rather than to sugar-coat – have further decreased pedagogical diversity by only teaching division.

*Sustainability*. On a related note, the system of educational interactions between the player and the game must inevitably collapse when the player successfully covers the material – e.g. acquires the knack for working simple arithmetic problems, or can easily defeat all of the zombies and solve all the refraction problems. Even if the player keeps playing, the educational experience has come to an end.

We now examine the games of chess and *StarCraft II*, which neatly solve the diversity and sustainability problems within their admittedly non-academic domains. Each game is surrounded by a large and growing body of (diverse) theory; and each game has shown incredible staying power (sustainability).

Their diversity yields an incredible amount of replay value (sustainability): every time one plays, the game takes a different path through an astronomically large possibility space. Being pitted against fellow creative human beings, in a tremendously complex domain, makes for a diverse range of playing styles, strategies, and in-game experiences[3]. This is in stark qualitative contrast with single-player games, in which the level of diversity is constrained by what can be designed upfront. Whereas the educational content of single-player games can eventually be depleted, the games of chess and *StarCraft II* have proven to be quite self-sustaining. We observe that this is because every competitor is implicitly incentivised to be a content producer as a byproduct of competitive play. In essence, each competitor crafts an experience for the other player – a kind of collaborative co-productivity in the midst of what is generally thought of as competition.

As we will demonstrate in the final section, the insights gained from studying chess and *StarCraft II* can be used to design personalized, authentic [BCD89, FP09] learning environments with high diversity and sustainability. We show this by discussing an environment centered around a 1-on-1 competitive game for teaching Java programming. The game's strategy revolves around crafting magic spells written in the Java language, and countering such spells crafted by other players.

[3]One might argue that in some games – like chess – playing against a computer affords the same benefit. This is true, but we would point out that it was a major technological achievement when Deep Blue beat Gary Kasparov. A lot of design work went in to making a strong chess playing computer. This supports our earlier claim that making an engaging single-player game is often a challenge, whereas in the competitive multiplayer setting, the engagement is often inherent

## 3.3 Related Work

## 3.4 Chess and StarCraft II

### 3.4.1 Why these games?

#### Diversity and Sustainability

We examine chess and *StarCraft II* because both games have exhibited extraordinary staying power (sustainability) and because both games have become surrounded by a rich body of theory to help players navigate diverse, in-game possibility spaces. Hence, they have established themselves as model domains for sustainability and diversity.

*StarCraft* is a real-time strategy game (RTS) which was released in 1998. Players assume control of a futuristic army and struggle for military dominance. Owing partly to the release of the game's expansion pack (*Brood War*), the *StarCraft* franchise quickly became a worldwide sensation – unofficially dubbed the "national sport" of South Korea. The site http://sc2ranks.com suggests that there are more than 4 million competitive players. Today, compared to other two-player competitive video games, *StarCraft II* consistently boasts the most tournaments, the most players across the globe, and the most spectators, making it one of the crown jewels of the growing phenomena known as "e-sports". The popularity of the world's number one strategy game has attracted researchers in sociology, anthropology, cognitive science, and artificial intelligence [CH11, KSC$^+$12, JML11, RW12]. Although it has been speculated that the game of *StarCraft II* and its community may have strong educational benefits [Bau12], we give the first-ever analysis of *StarCraft II* as a learning environment.

## Effective Learning Tools in Spite of Diversity and Sustainability

Both games have produced players that the learning sciences research community would consider “experts” in their field. Indeed, chess has long been considered a drosophila for studying human expertise [deg65, RCPS01, GC06], in order to draw conclusions about more socially relevant domains such as academic or professional expertise. This is because the domain complexity of chess and *StarCraft* begins to approximate the diversity and depth found in many academic fields. These games require incredible skill, precision, focus, and acquired knowledge. It has been estimated that grandmaster chess players make decisions based on knowledge of more than 50,000 familiar chess patterns [SC73a].

Not surprisingly, the games take years to master. Yet in spite of this, both games have consistently proven to promote sufficient motivation for players to invest the required “time on task” (for chess this can be from 10,000 to 50,000 hours [SC73b]). Furthermore, throughout this lengthy journey from novice to expert, the mechanism of 1-on-1 competition (against one’s peers) serves as a valuable feedback mechanism. One’s statistical likelihood of winning games against players of a given strength is a strong indicator of the degree to which one has internalized the necessary theory and skills. The Elo statistical model underpinning chess ratings is the standard for ranking and matching of chess players and rests upon very sound mathematics [CBP13]. All ranked players have a rating which ranges (roughly) from 0 to 3,000. As one becomes a stronger chess player, one’s rating improves. At all times, there is a quantifiable metric of one’s strength, allowing players to (for example) assess whether training regimens are working.

The robustness of this feedback mechanism in the face of such a diverse domain and sustained play is, we think, quite elegant. It is very difficult to imagine a single-player game that could sustain 50,000 hours of diverse play, while providing valuable feedback to anyone from novice to world-class expert.

**A Natural Research/Design Agenda**

For those wishing to design educational games that exhibit similar sustainability and which serve to teach something similarly diverse (such as an academic subject), one path is to:

- Analyse how chess and *StarCraft II* promote diversity and sustainability.
- Analyse what makes chess and *StarCraft II* such effective learning tools in spite of domain diversity.
- Design a similar game that interweaves academic content without sacrificing the above qualities.

We begin now with an investigation of the first two points regarding qualities of chess and *StarCraft II*. We finish the chapter by taking a stab at the third point.

### 3.4.2 Investigating chess and StarCraft II

The author of this thesis has been an active member of the chess and *StarCraft* communities for more than 10 years. The following analysis is based on a deep insider perspective, bolstered where appropriate by empirical research. For the gathering of empirical data, our methodology was the same for both chess and *StarCraft II*. We conducted a series of surveys on the relevant subreddits, first surveying 350 members of the *StarCraft II* community, and following up by surveying 40 members of the chess community (largely as a "sanity check" for the generalizability of the *StarCraft II* results). Although not this chapter's main contributions, many of the empirical results following from these surveys are novel contributions in themselves and can serve to inform future anthropological investigations of the *StarCraft II* community.

Our investigation was motivated by an interest in the following research questions:

- What is the nature of diversity and sustainability in these gaming domains, and what are the takeaways for game designers?

- What is the nature of 1-on-1 competitive feedback in relation to the above, and does such feedback effectively facilitate metacognition, positive self-theory, and domain knowledge acquisition for members of the community?

## Designing for Diversity: Balance is Key

In spite of the relative simplicity of chess's mechanics, its possibility space is combinatorially large. As such, a significant body of theory has been generated to serve as a guide through the space of possibilities. Within the body of theory, numerous open questions remain: Is it superior to occupy the center with pawns (the "classical" school of thought) or to control the center from a distance (the "hypermodern" school of thought)? Is it better for the first player to open by moving the King's Pawn or the Queen's Pawn? Does the Pirc Defence lead to a stronger position than the Modern Defence? Is the sacrifice of a pawn for an attack justified in the Queen's Gambit? To this day, such questions remain a matter of taste – largely aesthetic in nature. Differences of opinion exist in all echelons of chess expertise.

As it turns out, the body of theory surrounding *StarCraft II* exhibits a similar balance between differing ideologies. Of the three races[4], it remains an open question as to which is "best", with winning statistics being roughly equal for all three races (which is due, in large part, to Blizzard's rebalancing efforts – see below). Furthermore, when playing a single race, a variety of ideological approaches are available to players, with new ones being discovered and popularized constantly.

For example, consider a player who is playing as the "Terran" race (the human

[4]Instead of assuming command over one of two symmetric armies (white or black), a *StarCraft II* player can choose from one of three asymmetric "races" – each with their own unique soldiers, their own overall strengths and weaknesses.

race). She has numerous decisions to make in the opening few minutes of the game. She knows she must build a Barracks in order to train infantry soldiers. She can even build more than one Barracks in order to train soldiers faster. Doing so comes with an opportunity cost, however, because each Barracks requires expending in-game resources. Building an early second Barracks makes other options impossible – for example, a player cannot build a Factory as quickly. Because Factories are used to produce tanks, building two Barracks while one's opponent builds a Barracks and a Factory, might lead to a scenario wherein one has lots of early infantry, whereas one's opponent has fewer infantry but also has a tank.

There are also numerous other reorderings to the sequence of decisions one can make in the early minutes of the game. Many of these courses of action have been given names: e.g. the One-Rax Fast-Expand (building a single Barracks and expanding one's base immediately after); the One-One-One (building one Barracks, one Factory, and one Starport in rapid succession); etc. The point, however, is that a large number of these so-called "build orders" are viable. Having a surplus of infantry and no tanks against an opponent who does have tanks is hardly an immediate death sentence for either player. Players must struggle to position their chosen units where they will be most effective, choosing engagements carefully, and continuing to make decisions that reenforce one's strengths and repair one's weaknesses as the game unfolds. Having an army with a different composition from the opposing army gives the struggle an asymmetric nature – with each player employing a different style. Diverse gameplay is an immediate outgrowth of the coexistence of various, equally viable approaches to the game.

*Designing for Balance*. Blizzard's approach to encouraging balance in *StarCraft II* is to carefully monitor data acquired from all games of *StarCraft II* played online. Blizzard periodically "rebalances" the game by releasing patches that tweak the relative value of certain units in the game – i.e. increasing the cost of a particular soldier, or decreasing the power or range of certain attacks, etc. This alters the competitive advantage of some races over others and some styles of play over others. By enforcing balance within the game, diverse gameplay is maintained.

In the final section of this chapter, we show how we designed our own competitive possibility space – in which players craft offensive and defensive magic spells by writing Java code. Our user study was conducted over a long period of time precisely to determine whether the resulting body of theory that arose around the game exhibited features of balance between various approaches to the game. We recommend a similar empirical investigation to other designers interested in crafting educational games within the 1-on-1 gaming paradigm.

## Assessment and Feedback through Competition

Having mentioned prior work on the importance of feedback, we now consider feedback from a game designer's perspective. In a single-player game, the burden of assessing the player's performance and providing feedback must be carried by the environment. This means that, in turn, the burden of designing assessment mechanisms and feedback mechanisms falls to the designer. In a game like *Math Blaster*, positive feedback is given when the player successfully solves math problems. Similarly, *Zombie Division*, players successfully kill zombies when they choose the weapon with the correct divisor. The diversity of content in single-player educational games is, therefore, partially constrained by what kinds of player activities can be checked and reported upon during runtime.

By comparison, the feedback mechanism in 1-on-1 competition is provided not by an algorithm, but by one's opponents. If a player consistently wins against a particular opponent, this is a strong indicator that the winner has a superior, more expert-level grasp of the game's body of theory[5]. The crowd-sourced nature of feedback in 1-on-1 competitive games has unique drawbacks and benefits that should be understood by designers.

*Drawbacks*. Winning (or losing) a competitive game is a form of feedback. From a learning sciences perspective, the main drawback of such feedback is that a win/loss result is a purely digital, 1-bit judgement over a very complex cognitive task. A player does not necessarily know if she won because her king-side pawnstorm was a good idea; or because the opponent failed to make a timely counter-punch in the center; or because she elected to exchange her light-squared bishop to double the opponent's pawns; etc. Likewise, in *StarCraft* it is not always immediately clear what factors of good play, bad play, and/or randomness most contributed to the outcome. Thus, it is not always clear what weaknesses or misconceptions a learner must target for future training.

*Benefits*. Ascertaining the "truth" of a win/loss result can require significant analysis. Although this can be a drawback, it is this very drawback that shapes some of the practices that surround these games. It is generally accepted etiquette at chess tournaments, for example, for opponents to leave the tournament hall in the wake of a finished chess game and to discuss it privately with each other – a procedure known as the "post-mortem". Moreover, players of all levels study their own games, sometimes with the help of peers, stronger players, or computerized analysis to identify their own weaknesses. This is just as critical in *StarCraft II*, where (in contrast to chess) one does not have perfect information about what the opponent is doing during the game. Because one cannot see the opposing army except when it is near one's own units, it is quite common, especially for novice players, for an onslaught to "come out of nowhere", taking a player by surprise and leading to defeat. Examining the game's replay (wherein all troop movements are revealed) is necessary in order to determine how such a *blitzkrieg*

[5]This is true because it is essentially a tautology: The game's body of theory is precisely the knowledge that, when mastered, leads to more consistent victories.

was possible, and to prevent it from happening in the future. The act of analysing one's own games and the games of others in order to make sense of win/loss result is a powerful form of metacognition, which can be extremely beneficial. We examine metacognition and the relevant metacognitive practices of the community in more detail momentarily.

Another benefit is that the aggregate of 1-bit feedback results, over time, begins to paint a more meaningful picture of one's playing strength. This has been mathematically formalized with the so-called Elo rating system, which assigns to players a numerical value called a "rating". The Elo rating system is a statistical model that undergirds the selective matchmaking algorithms used at chess tournaments and on chess servers to automatically pair up opponents who have similar ratings[6]. Today, the same or similar statistical models have been applied to a variety of competitive games – from Go, to soccer, to baseball. A player's rating relative to other players in the community is a very accurate indication of that player's strength. The more a player plays, the more statistically reliable the rating becomes.

Although Blizzard does not make its exact matchmaking algorithm public, it does seek to match players of similar skill, thus increasing the probability that each game is a challenge for both players. And although Blizzard does not reveal to players their exact numerical rating, Blizzard does categorize players into "leagues" (Bronze, Silver, Gold, Platinum, Diamond, Master, and Grandmaster) and does rank players within these leagues. Just as chess players ratings can increase and decrease over time (due to iterated wins and losses), a *StarCraft II* player can be promoted and demoted to different leagues as a result of changes to one's hidden rating. The result is that players can chart their improvements over time, allowing them to obtain a general picture of how effective their training regimens have been. We expound upon the nature of these training regimens momentarily.

[6]If two players have the same rating, they each have a 50% chance of winning.

Finally, another benefit is that 1-on-1 competition is a feedback mechanism that scales well in the face of domain complexity. This is because the feedback mechanism does not hinge upon a designer's ability to embed algorithmic feedback into the game environment. Designers need only construct an environment that can algorithmically detect when a game is won or lost. As we have alluded to, the body of theory surrounding both games is large, growing, and difficult to master. The road to domain expertise is a long one. Yet chess players were able to acquire meaningful, accurate performance feedback in the midst of these complexities long before Deep Blue came along and made automated performance feedback a reality. Likewise, even with the "Deep Blue" of *StarCraft II* still far off, obtaining accurate feedback over the full range of one's *StarCraft II* playing abilities is easy, due to the crowd-sourced nature of competitive feedback from one's approximate peers. Thus, 1-on-1 competition is a tool that designers can potentially leverage when automated feedback is difficult or impossible to implement.

**Training Regimens**

With a complex domain requiring mastery and a 1-bit feedback mechanism at one's disposal, we wanted to investigate how players undertake the journey from novice to expert. Given the popularity of video games, it is no surprise that our surveys found that chess and *StarCraft II* players spend a considerable amount of time playing their game of choice. What has thus far *not* been observed in research literature, however, is that players also spend a significant amount of time on supplemental training.

Chess players engaged in 8.3 hours of practice on average per week, with about 20% of this devoted to supplemental training (working chess puzzles, reading chess books, watching chess online, or analysing chess games). Likewise, *StarCraft II* players reported spending on average 11.6 hours per week practicing *StarCraft II*, with about 27% of this time devoted to supplemental training – e.g. sparring with team mates,

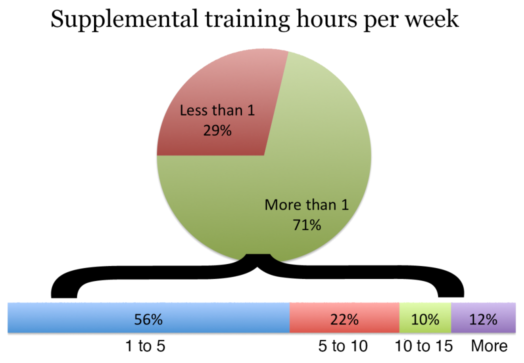


**Figure 3.3**: Results of a survey investigating *StarCraft II* players' training habits – *aside* from merely playing *StarCraft II*.

watching live streams, watching web shows about *StarCraft II* strategy, analysing the games of strong players, or taking lessons. Figure 3.3 summarizes the average hours spent on supplemental training (over and beyond just playing *StarCraft II*).

Not only are these empirical results novel, but we think they may come as a surprise to some members of the research community. In our experience, video game researchers seem to realize that gamers *play* a lot – but not that gamers invest a considerable amount of time *training*. Chess and *StarCraft II* players realize that "all play and no work" is not the key to success.

## Metacognition: Analysing Games

A large component of the training regimens observed in the previous section involves the performance of metacognitive acts – e.g. analysing one's games after the fact. We also consider analysing the games of others (e.g. grandmasters) to be an important metacognitive act: when doing so, a player cannot help but perceive the grandmaster's game through the lens of her current level of understanding, noticing decisions made by the other player which are the same or different from the ones she would have made in the same situation. An awareness of the surprising features of a stronger player's game

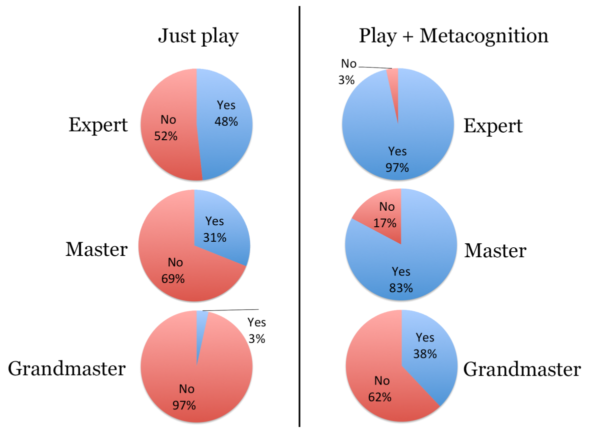


**Figure 3.4**: Results of a survey investigating chess players' perceptions of how proficient one can get with and without the benefit of metacognitive activities. For example: 48% of those surveyed believed that the level of Expert could be achieved by *only* playing chess

is actually an awareness of one's own potential shortcomings; and an awareness of the features which seem "obvious" is actually an awareness of one's own strengths.

We surveyed the community to investigate the perceived importance of analysing games – either one's own games or the games of others. Of the players we surveyed, 82% of chess players and 71% *StarCraft II* players said that they spent the majority of their supplemental training time engaged in such activities – for example: analysing their own games, studying grandmaster replays, watching web shows about replays, or watching games being streamed live. Indeed, Figures 3.4 and 3.5 summarize players' perceptions regarding whether one can reach high levels of play *without* engaging in some kind of metacognitive activity – over and beyond simply playing games. Members of both communities clearly consider metacognitive activities to be highly beneficial (or necessary) for reaching upper performance echelons.

The site chessgames.com makes over 650,000 chess game records available over the internet. Chess games played in most major tournaments become part of the public domain, thus adding to the instructional content available to the community. Weaker players routinely study these games in order to sharpen their own skills.

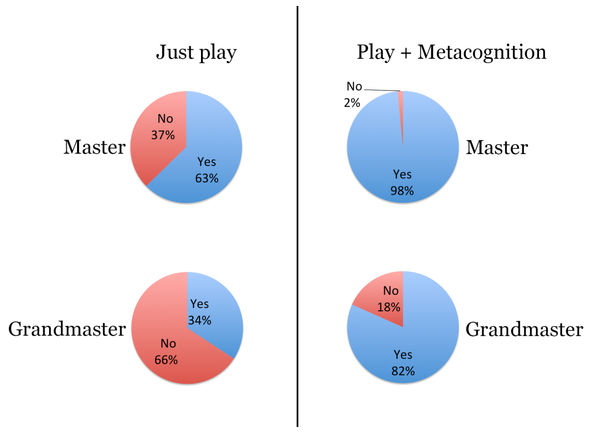


**Figure 3.5**: Results of a survey investigating *StarCraft II* players' perceptions of how proficient one can get with and without the benefit of metacognitive activities. For example: 34% of those surveyed believed that the level of Grandmaster could be achieved by *only* playing *StarCraft II*.

*StarCraft II* games are recorded automatically and can be streamed live via screen capture. Like records of chess games, *StarCraft II* replays serve a vital community function. The site gamereplays.org contains 65,000 uploaded replays. Games by top players are routinely downloaded more than 5,000 times within just days of being posted. It is not uncommon for replays of championship matches by world-class players to be downloaded more than 10,000 times [gam12]. A popular web-show *Day 9 Daily* (viewed live by thousands) centers around high-level commentary of replays. And thousands of people tune in online to watch players of various strengths stream their games live [KSC$^+$12]. This phenomenon is a sustainable practice of the community largely because strong players enjoy playing, and because many weaker players want to play like strong players. (Moreover, popular players can be further incentivised by sites like Twitch.tv, which apportion a cut of their advertising revenue to streamers.)

## Positive Self-Theory

The road from novice to expert in chess or *StarCraft II* is fraught with continual failure. And by design, failure is a daily occurrence: players are automatically matched

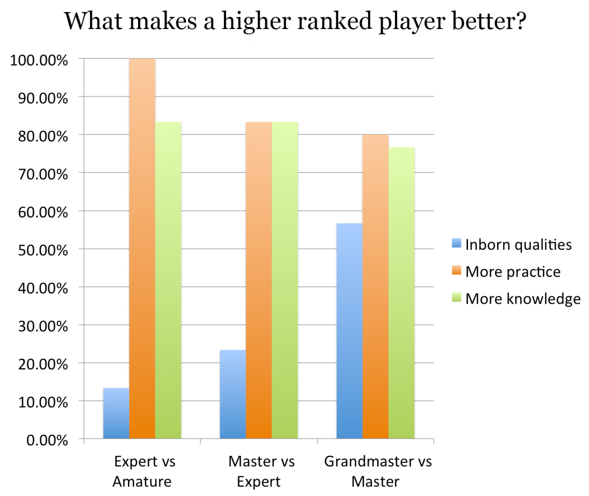


**Figure 3.6**: Results of a survey investigating chess players' perceptions of what makes a higher ranked player better than those in a slightly lower-ranked category. For example: 100% of those surveyed believed that experts are better than amateurs due (in part) to having practiced more.

with other players who have a similar rating, meaning that, even for good players, losses are as common as wins. One could imagine that this aggregate of losses could begin to instill negative self-theory.

We surveyed the community to investigate this potential drawback. In particular, we asked the gaming communities a series of questions about why certain players perform better than others. Following in Dweck's theoretical framework, we were mainly interested in whether players believed that a predisposition to certain cognitive skills was inborn (which would indicate a fixed-mindset) or whether these skills could be improved (which would indicate a growth-mindset) – or some combination of these. Figure 3.6 summarizes our findings for chess. As can be seen from Figure 3.7, the trend with regard to *StarCraft II* is very similar.

In both communities, players believed that, at all levels of play, a combination of 1) a greater amount of practice and 2) more previously acquired knowledge were the primary causes of better performance. In-born talent was viewed as a possible deciding

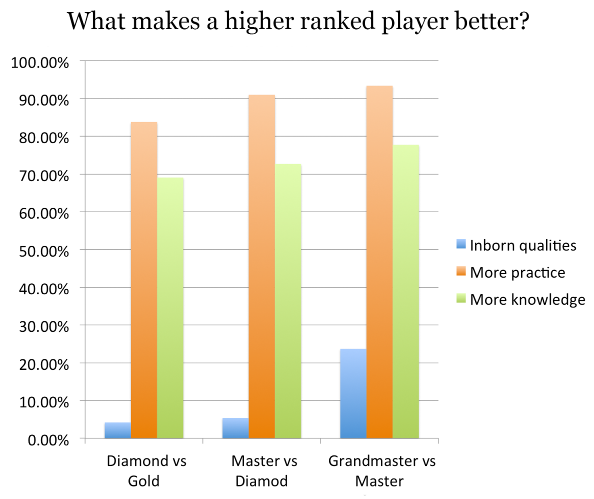


**Figure 3.7**: Results of a survey investigating SC2 players' perceptions of what makes a higher ranked player better than those in a slightly lower-ranked category. For example: about 25% of those surveyed believed that Grandmasters are better than Masters due (in part) to inborn advantages.

factor only at the highest echelons of skill. Since the average player (and 100% of those surveyed) are not members of these echelons, it is safe to say that the average player views her own skill or lack thereof as a non-fixed, changeable, "growable" quantity. This mindset is especially pronounced in the *StarCraft II* community. Only 5% of the community believes that inborn qualities have a bearing on amateur-level competition. And since masters and grandmasters comprise only the top 2% of all *StarCraft II* players [bat12], amateur-level competition is the bulk of the iceberg in a community of millions.

However, we must be cautious about drawing the conclusion that 1-on-1 competition somehow *instills* a positive self-theory. We can only conclude that those who elect to join and remain members of these communities *have* a positive self-theory. Naturally, we could not survey players who may have played for a while, become discouraged, and quit forever. It is possible that these games may attract or retain players who are more disposed to having a positive self-theory in the first place.

## 3.5 Theory into Practice

### 3.5.1 Overview

The salient features of the design, implementation, and evaluation of our game – *CompetitiveSourcery* – flow directly from the insights gleaned from studying the 1-on-1 competitive games of chess and *StarCraft II*.

- *A large possibility space.* Players can write arbitrary Java code in order to craft magic spells to spawn and control various in-game 3D objects within a simulated-analog 3D world.
- *Competitive Feedback.* Spells are designed to slay or entrap opponents. Players compete in iterated 1-on-1 competitions, pitting their spells against the spells of other players.
- *A long-term study.* We studied the dynamics of our game in the long term in order to determine whether the resulting body of theory (if any) which arose around the game exhibited the kind of balance that correlates with diversity in chess and *StarCraft II* (see section 3.5.3).
- *A small group of users.* We did our study on a small group of users in order to debunk the potential misconception that a diverse body of theory is only possible when a community of millions is exploring the possibility space.

### 3.5.2 Design and Implementation

To implement the multiplayer version of CodeSpells, we leveraged the preexisting *CodeSpells* platform [cod12] – a single player, non-competitive game. Although the game is for novices, it has an extensible underlying framework allowing us to implement networked multi-player support that *CodeSpells* lacks and to build a new 1-on-1 competitive game around the notion of "dueling wizards". The interface features an in-game IDE through which spells can be written. It also features a powerful spell-crafting Java API for manipulating the 3D environment around the spell-caster.

The rules of the multiplayer game we designed are relatively simple. One player assumes the role of the red wizard, and the other the role of the blue wizard. Both wizards appear in an outdoor arena, where they attempt to eliminate each other. Competition in the game revolves around attempting to cause collisions between one's opponent and wooden boxes of one's own color. Each such collision reduces the health of the wizard in question. The first player to lose all of her health loses the game. Since red boxes will not damage the red wizard (and blue boxes will not damage the blue wizard), one's boxes may be assembled into structures such as towers and platforms which are hostile to the opponent but safe to oneself. Importantly, boxes may only be created and moved via code crafted at runtime inside the game's IDE.

There was one other critical aspect of our study which was not explicitly designed into the game: the subjects in our user study understood that they were members of a three-person intramural team that would represent their department in an upcoming tournament. Partly inspired by recent investigations of team practice environments [DKP12], we felt that engendering a team mindset would help contextualize all inter-subject competition as friendly collaboration. Games played between subjects were, on the one hand competitive (with an objective winner and a loser), but on the other hand were contextualized as making the team stronger. This allowed us to preclude the various negative emotions that can be associated with 1-on-1 competition.

### 3.5.3 Evaluation

We were mainly interested in whether the game would turn out to be nuanced enough to inspire a large body of theory to arise around it[7]. This relates to a few of the main goals of this thesis: 1) build coding games that involve things that we know gamers like, and 2) build games that incentivise engagement with a range of CS1 concepts. To the first point, we know that gamers like strategy games like chess and StarCraft II; and we showed earlier that gamers spend a lot of time engaging with the bodies of theory around these games. To the second point: We hypothesized that engaging with the body of theory around multiplayer CodeSpells would imply engaging with a range of CS1 concepts.

We chose to do an in-depth study with three users. All were 2nd year CS majors who had taken at least one programming course. One was male, the other two were female. The study lasted two months. The users gathered together three or more times per week for two hours at a time to play. Over all, this totalled to almost 150 person-hours of time devoted to playing and/or to metacognitive activities thereupon. The content of these two-hour sessions was not dictated by the researchers and was left to the discretion of the three-person team.

We observed these sessions and took detailed field notes, regardless of what team-related activities the subjects chose to engage in: e.g. playing games, discussing strategies, debugging each other's code, or updating the team wiki.

#### Diversity and Sustainability

The first week of the study saw the rise and fall of a crude strategy which initially appeared to be unstoppable: Players would write an infinite loop that spawned a new

[7]Arguably many games – even simple ones like tic-tac-toe have a body of theory. However, with games like chess and StarCraft II, the body of theory is much deeper – which we believe correlates with the game's replay value.

box at some arbitrary location L on every iteration. Because this resulted in a multitude of boxes being spawned on top of each other within a short period of time, the game's built-in physics engine would cause the boxes to fly away from the origin point L in non-deterministic directions as if they had collided at a high speed. This strategy was named the "Bomb" and prevailed for about two days – bringing consistent victory to whomever employed it. The Bomb strategy's high level of efficacy virtually forced both players to mutually employ the *same* strategy, at which point the gameplay was characterized by each player trying to avoid the other player's bombs while casting bomb spells of their own. In this case, winners were generally determined by who could best outmaneuver the opponent. Also, many games ended in a draw – with both players having spawned more boxes than the game's imposed limit (about 100).

At first, we were worried that a tipping point had been reached and that players would no longer be encouraged to write new Java programs. However, in short order, one of the subjects discovered that she could write a spell to pick up and carry a box by making a loop that would continually set the location of the box relative to the position of her avatar. In this way, she could carry the box directly to her opponent in order to inflict damage. This was dubbed the "Battering Ram" and immediately made the Bomb obsolete – owing to the fact that the Battering Ram was much more precise and did not suffer from the 100-box limit.

Gameplay then began to take on a fencing-esque character – with players trying to maneuver in such a way that they could tag their opponent with their battering ram without being tagged in turn. However, the strategic arms race escalated further with the introduction of the first viable *defensive* strategy: One of the subjects discovered that he could write a spell to levitate a crate of his own color and move it along his avatar's forward vector – the utility being that (if he was standing on the box), he could fly. While flying, he could inflict damage on his opponent by using the Bomb strategy, flying overhead and spawning crates at his own location, causing them to bombard his opponent from above. This was dubbed the “Fokker”[8]. Using the Fokker, a player could easily defend against the earth-bound Battering Ram, countering it effectively from the skies.

Two counter-strategies arose next: the Fokker Battering Ram (a combination of the Battering Ram and the Fokker), and the Gun. The Gun was a spell that would spawn a box at the avatar’s location and launch it forward, creating a dangerous projectile. The Gun was first employed as a ground-based anti-aircraft strategy to shoot down players employing the Fokker. However, it was later combined with the Fokker to create the Fokker Gun strategy, which (along with the Fokker Battering Ram) could be used to do air-to-air combat against someone employing any of the aforementioned Fokker-based strategies.

This was less than one month into the study, and the pool of viable strategies was quickly growing. This divergence, in and of itself, had a noticeable effect on the characteristics of strategies that came later: Because it was impossible to know beforehand what strategy one’s opponent would employ, the best strategies were the ones that were most flexible in the face of uncertainty. Players would, for example, open the game by flying (i.e. the Fokker), but would delay the dropping of bombs until the opponent had committed to a ground-based or aerial counter strategy. In the case of a ground-based counter strategy, the traditional Fokker (i.e. with bombs) appeared was quite effective. In the case of an aerial counter strategy, the Fokker Gun or the Fokker Battering Ram was better suited for air-to-air combat. In short, the growing pool of strategies necessitated that new strategies be more complex to cope with the growing uncertainty.

In the interest of length, we cannot go into detail about every strategy developed

[8]An allusion to a World War I German aircraft – one of the earliest examples of a plane that could fly while dispatching machine gun fire.

during the course of the study. The team wiki page describes over 30 distinct strategies, some requiring the user to correctly write and execute three or more distinct spells in-game and to accurately adapt to what the opponent is doing.

All in all, we were pleased to see a divergence of effective strategies – rather than a convergence toward a single dominant strategy. We are pleased to report that, to date, after 150 man hours have been spent in search of a dominant winning strategy, no such strategy has surfaced.

Furthermore, the arms race continues to this day. At this time, our subjects are developing a style of play based on teleportation – where the player teleports from place to place in an attempt to confuse the enemy. Other promising strategies have intriguing architectural characteristics: the building of walls, bridges, stairwells, ceilings, and moving platforms.

## Pedagogical Merit

Having shown that the set of viable strategies exhibits signs of increasing diversity, it is still worth asking how diverse the academic skills being sharpened are. If it should happen that some of the strongest strategies involve Java programs that can be written without loops or branches, for example, then the pedagogical benefits could be jeopardized due to insufficient content coverage.

*Coding Constructs.* We can report, though, that even the most basic of spells written during our study tended to draw from the entire bag of first-year programming constructs: if-statements, loops, boolean expressions, arrays, etc 3.8. And the more powerful spells almost always tended to be the more flexible ones – i.e. the ones with more control structures. Also, each individual spell is structured as a Java class, which introduces a need for object oriented programming, particularly in cases when two or more spells must work together – which became common as strategies became complex.

| | Expr | Func | Contr | Data | OOP |
|---|---|---|---|---|---|
| ram | 0 | 9 | 1 | 5 | 1 |
| portal | 0 | 6 | 2 | 3 | 1 |
| platform | 7 | 19 | 4 | 7 | 1 |
| rain | 1 | 1 | 1 | 4 | 1 |
| bridges | 1 | 8 | 1 | 5 | 1 |

**Figure 3.8**: A graph of concepts detected during corpus analysis of 5 of commonly used programs. Each column maps to a CS1 concept (see Chapter 2): Expressions, Functions, Control, Data Types, Object-Oriented Programming. The five rows map to five programs in the strategy space. Cells contain points that indicate the number of CS1 coding constructs appearing in the program. Points were calculated as follows. Expressions: 1 point for any non-trivial math or boolean expression (trivial being a single token); Functions: 1 point for any function call or definition; Control: 1 point for any if-statement or loop; Data: 1 point for each distinct data type; OOP: 1 point for each class written.

*Software Engineering.* At an even higher level: The software engineering concept of "modularity" arose quite naturally after less than one month of the study. Our subjects realized that using many small spells to execute a complex strategy was more flexible than using a single monolithic spell. For example, the spell to fly – one of the two spells involved in the basic Fokker – was kept separate from the spell to bombard the opponent from above, because this allowed the basic Fokker to be easily switched to the Fokker Gun or the Fokker Battering Ram as necessary.

Furthermore, because the space of known strategies was in constant flux, the subjects had to understand, evaluate, and design against ever-shifting requirements – a skill set that (in our experience) can help software engineers keep pace in the modern tech industry climate. They had to maintain their code and its documentation when innovative new counter strategies were discovered. Refactoring and software maintainance were common activities.

*Human Factors.* Subjects were also forced to wrestle with tough HCI challenges – e.g. figuring out how to make powerful strategies more straightforward to execute in-game. For example, one subject discovered that she could write spells that would stall (in a loop) until she pressed the spacebar to make her avatar jump. Thus, after casting more than one of these “delayed spells” in a safe location, she could activate all spells at once with a single button press at an opportune moment. This was much easier than casting one spell after another during the heat of battle.

The aforementioned breadth of academic content is worth comparing to traditional single-player educational games – where the range of academic material is limited to narrow domains in which algorithmic feedback can be provided (e.g. arithmetic or simple division). By contrast, our subjects were exposed to academic diversity spanning from first-year programming skills all the way to fourth-year skills and beyond.

The team practice environment bears more resemblance to that of a startup company than a college classroom – which, from a learning sciences standpoint, is a positive quality. Students are known to learn better when concepts are encountered in a meaningful context, rather than as disembodied abstractions [BP02]. Many researchers have also called for a greater level of authenticity in education – where learning environments are crafted to reflect the “authentic” [BCD89, FP09] activities of the relevant “community of practice” [Wen99]. We believe the team practice environment centered around the game qualifies as “thickly authentic” [Sha04, Sha05], capturing an unabstracted mix of the hard and soft skills that real computer scientists use every day.

**Metacognition**

The team of subjects tended to follow each game by doing a post-mortem, in which the subjects were frequently observed saying things like, “I lost because I need to write that spell faster next time. I couldn’t remember it well enough.” or “I messed up the loop, so the spell didn’t work. That was silly.” These are compelling examples of how the feedback provided by lost games serves to highlight weaknesses in the player’s coding abilities. In the wake of these post-mortems, subjects proceeded to strengthen these self-identified weaknesses through further drilling and practice.

We also observed a different sort of discussion during the post-mortems – e.g. "I don't think strategy X is working well against strategy Y. What if I were to tweak X by doing Z?" From a pedagogical perspective, these responses to losing a game are undeniably encouraging. They are instances of creative problem-solving, and they routinely emerged as a result of the win-loss feedback provided by competing with one's team members. Many of the novel Java spells implemented by the team were devised in this way – as changes to existing code, prompted by a new understanding of the requirements arising from the analysis of a competitive result.

During analysis, subjects spent a considerable amount of time discussing and deriving new creative strategies. Indeed, as much time was spent on these activities as on playing the game itself. As mentioned previously, bouts of creativity tended to occur when reflecting on why some strategy did not perform as well as expected. These metacognitive acts no doubt have pedagogical benefits as well – sharpening the so-called "soft" skills of being able to talk intelligently about code and to solve problems collaboratively.

### 3.5.4 Future Work

Other issues aside from metacognition and mindset are also critically important in STEM, especially for the recruitment and retention of women – for example, self-efficacy [JS05]. Self-efficacy (which is perhaps related to mindset) pertains to feelings an individual has about whether a field is "for them" and/or whether they can succeed in the field. An interesting avenue for future work would be to examine how gaming communities (competitive or otherwise) have or don't have similar problems with self-efficacy or lack of role models. If possible, it would be beneficial to study marginalized groups and outsiders in the gaming community. There are almost as many female gamers as there are male gamers [esa15], so a deeper study of how various communities include or don't include women could be leveraged to inform the design of coding games such that they are maximally inclusive.

Additionally, as we noted above *bouts of creativity tended to occur when reflecting on why some strategy did not perform as well as expected.* We think this is an interesting result: that creativity may be sparked by failure. Are we more metacognitive in the face of failure than victory? We would be interested to know if this has been studied in other domains. Additionally, it is possible that designers could attempt to design for such experiences to occur. One example would be, after a player experiences a loss, the game immediately places the player into a "post-mortem" interface, where the game can be analyzed. Perhaps the game could also statically analyze the code that both players used and suggest differences that may be indicative of why one player lost and the other won. StarCraft II does something similiar, in that when the game completes, the player sees a screen where the game can be replayed and various statistics can be analyzed. It would be intersting to consider how such an interface might work in the context of a coding game.

## 3.6 Conclusion

In our prior work on CodeSpells, we have focused on singleplayer gameplay. Other coding games, such as Lightbot have also done this [lig12]. And we know of now currently available multiplayer coding games. But there is tremendous potential in multiplayer. CodeSpells is a great example of this: consider the tradeoffs of quests vs multiplayer – two ways of incentivising coding. After months of non-trivial design work [EWF$^+$14], we were able to create a singleplayer quest-based experience that could provide up to 10 hours of diverse gameplay for novice elementary schoolers [EFG$^+$14].

The number would be *even less* for a more experienced coder who could complete the finite number of quests faster. But by designing an simple multiplayer game (which took a few hours), and building a small community around it, the resulting system provided over 50 hours of diverse gameplay for intermediate coders, with no apparent upper bound. They could have continued exploring the strategy space, crafting more and more intricate strategies. This allowed us to design for higher replay value at lower cost. For this reason, we recommend that other designers leverage the power of competitive multiplayer in coding games. The results of our long term study indicate that our environment is both sustainable and diverse, inspiring high levels of content coverage, engagement, and metacognition.

## 3.7 Acknowledgements

This chapter uses previously published materials:

- Stephen R. Foster, Sarah Esper, and William G. Griswold. 2013. 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, 99-108.

- Sarah Esper, Stephen R. Foster, and William G. Griswold. 2013. CodeSpells: embodying the metaphor of wizardry for programming. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education* (ITiCSE '13). ACM, New York, NY, USA, 249-254.

# Chapter 4

# Programming By Gaming: Coding as a Game

## 4.1 Introduction

A recent trend in educational game design and research goes by many names: seamless integration [FEG13], immersive didactics [MWM12a], immersive learning [AVW07], learning-gameplay integration [MM11], intrinsic integration [HA11], embedded learning [BBGP09], stealth learning [Pre04], and avoiding "chocolate-covered broccoli" [LKLC11]. 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 [HA11].

In this chapter, we focus on a particular kind of coding game that is also a visual programming language. Key take-aways are 1) the design of the game itself and how we encoded coding constructs (e.g. list transformations, recursion, conditionals) in

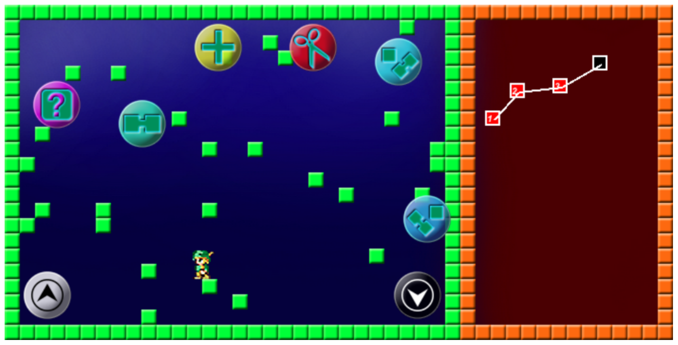


**Figure 4.1**: On its surface, The Orb Game is a 2D Mario-like game. The avatar (in the lower middle) can trigger various ”orbs” throughout the environment. Doing so will transform the data displayed in the right-hand panel in various ways. In this game, users think they are playing a game and solving concrete puzzles. However, they are actually writing programs that operate on abstract inputs.

a game, and 2) the overarching design process for building a Turing-complete visual programming language out of game mechanics – a process that could be repeated for other genres of games. 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 have an embedded IDE (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 chapter 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.

## 4.2 Fully Seamless Design

### 4.2.1 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.

### 4.2.2 Related work

*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 [CHK$^+$93] and Tinker [LH80]. 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 [FEG13], and CodeSpells [EFG13a]. 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.

### 4.2.3 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 4.1. 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 4.1, 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 4.1.

- *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 4.1, 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 4.1.

- *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 4.1.

- *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 4.2.2.

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 touches the red orb again – the "pop" function. The element 2 becomes separated from the list. See Figure 4.2.3. 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 4.2.4. 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 4.2.5. The *add* box now produces 6. See Figure 4.2.5. 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.

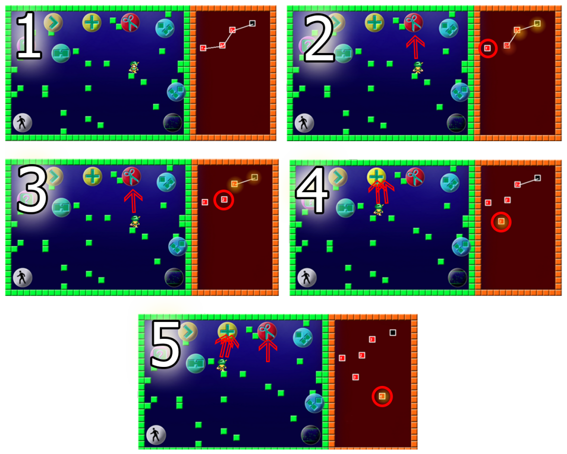


**Figure 4.2**: 1) Bob begins with the list [1,2,3]. 2) Bob triggers the pop action, popping the first element from the list. 3) Bob pops the second element from the list. 4) Bob triggers the add action twice, adding 1 and 2, producing 3. Finally, 5) Bob does another pop and another add, producing a 6.

*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 4.3.2. 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 4.3.3. 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

[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.

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, 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
```

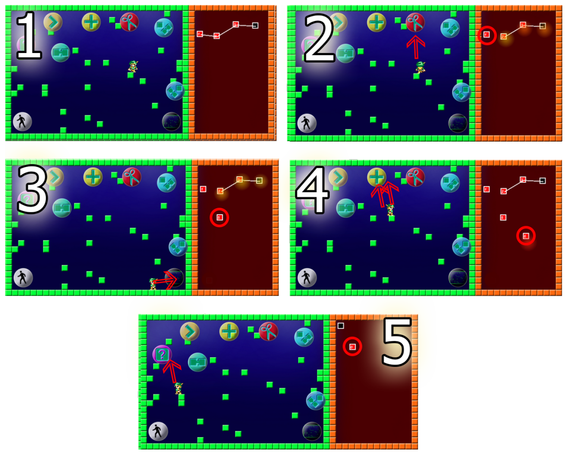


**Figure 4.3**: 1) Bob begins with the list [1,2,3]. 2) Bob triggers the pop action, popping the first element from the list. 3) Bob triggers the recursion orb; the oracle returns 5. 4) Bob triggers the add action twice, adding 1 and 5, producing 6, which he returns. Finally, 5) Bob must solve the base case, which he does by activating the define constant orb, getting a 0, and returning.

```
    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 4.4 for an image of the stacked game instances.

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

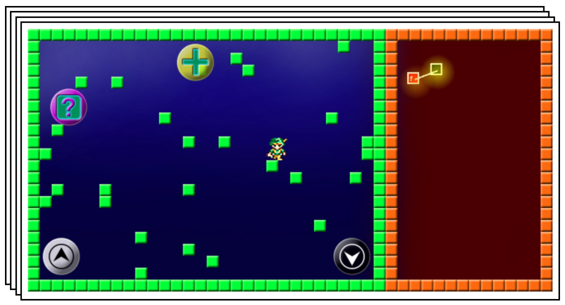


**Figure 4.4**: The runtime stack visualized as an actual stack of games during execution. When the avatar triggers the return orb in the lower left, the top-most level will be removed, and the currently selected items in the inventory (the list [1]) will be returned to the next game in the stack.

- 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?

## 4.3 Method

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 a 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.

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.

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

| *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 4.5**: 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.

anything would make the game more fun? What (in your own words) does the black orb do?

## 4.4 Results

The results of each subject's performance on each benchmark are contained in the table in Figure 4.5. 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 4.5 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 [JS05]. 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.

## 4.5 Discussion

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 [vAD04]. Pipe Jam [DDE+12], 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.

## 4.6 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.

## 4.7 Acknowledgements

This chapter uses the following previously published material: Stephen R. Foster, Sorin Lerner, and William G. Griswold. 2015. Seamless Integration of Coding and Gameplay: Writing Code Without Knowing it. In *Proceedings of Foundations of Digital Games* (FDG '15). New York, NY, USA, 3063-3072.

# Chapter 5

# G(ATS): Coding for a Game

## 5.1 Introduction

Game-based education is a field of research that has spawned hundreds of publications, systems, and games over the last 10 years [ENS06, Mac11, GM08, GmGmGc04a, MTJV09, LK11, FCJ04, IFH10]. This is no surprise; the holy grail of game-based education is an enticing one indeed. Commercial video games inspire players to spend countless hours "on task", while teachers struggle to keep students on task with traditional educational techniques. Perhaps using games and gamification, students can be inspired to learn academic subjects too.

The particular academic subject we are concerned with in this chapter is computer science – or more specifically, coding. Coding education has been recognized time and time again as a critical field of research, largely due to the massive gap between the number of computer science college and the number of computer science jobs that will be created in the next decade [cod14]. When it comes to game-based education for novice coders, the early results are encouraging. There have been several notable successes both commercially and in laboratory settings. We review some of these successes in our related work section.

This chapter seeks to fill two gaps in existing literature on game-based coding education:

- The literature does not evaluate a coding education system integrated with a blockbuster game. Because of this literature gap, the research community does not know what happens when educational content is embedded into a game so compelling that students would rather play it than learn from it. We examine this in 5.3 and 5.5. We uncover significant challenges when the game is “too fun”, and we present pedagogical solutions: e.g., how to sufficiently gamify the coding aspect of the experience.
- The literature does not compare game-based coding education to the state-of-the-art in *non*-game-based coding education. In 5.7, we compare a game-based education system with a non-game-based education system. The results are surprisingly asymmetrical: neither is clearly superior. As such, we must recommend a hybrid approach that can take advantage of the best of both. Namely, Scratch (or a similar system) should be used to reduce cognitive load while teaching concepts, whereas Minecraft (or similar game) may be used as a playground to make coding relevant to students.

We also put to rest a few tempting myths that we’ve encountered: e.g., 1) games will magically motivate students to push beyond the inherent difficulties of coding; 2) more gamification monotonically equates to higher engagement and/or better learning; and 3) the more fun the game is, the more a student will enjoy the learning experience.

Game-based education has tremendous potential. But our results show that it is not an educational wonder drug.

## 5.2 Related Work

Software systems for teaching coding are myriad. Rather than giving an exhaustive list of them, we give a taxonomy of gamification tactics we've seen used in coding education.

- *Traditional Approaches.* Traditional approaches to teaching coding use no gamification (unless one is willing to classify quizzes, tests, letter grades, etc. as gamification). An example of this apporach might be a typical "Introduction to Java" class. Students write code in IDEs like Eclipse or BlueJ. Their code prints strings to the console and/or manipulates rich media [FG04]. We are not here to comment on the effectiveness of this approach. We simply observe that this is the baseline – the most common interface for teaching coding.

- *Building games.* In some classes and online e-learning systems (e.g. Codecademy and Khan Academy) students design and/or build games with code. This technique is growing in popularity and have been part of the research literature for years [Rit09, PCR11]. The idea behind this strategy is simple: Students enjoy playing games; perhaps they will enjoy making them too.

- *Game-like IDEs.* Coding environments can be designed to look and feel like games, as demonstrated by several state-of-the-art educational tools: Alice [DCP06], Scratch [MRR$^+$10], NetLogo [Dic11], Kodu [Mac11], Greenfoot [KÏ0], etc. These systems do away with text-based code, using draggable blocks instead. The overall experience looks and feels more like a game than writing in, say, Eclipse. Other gamification techniques can be layered on top of this general UI paradigm; for example, commercial software called Tynker motivates students by giving them badges to unlock while they write code in a Scratch-like IDE.

It should be noted that a gamified IDE can be combined with the aforementioned *building games* technique to further increase the level to which games permeate the learning experience. For example, it is common and effective to teach game development using Scratch [Ke14]. Indeed, the authors have taught courses like this for years to local K-12 students.

- *Extending or modding an existing game*. Commercial titles like Skyrim, Minecraft, and Garry's Mod are all games that can be "modded" or extended by writing code. These games, along with their modding environments, have the potential to be playgrounds for learning to code [ENS06]. In spite of the fact that writing mods is difficult, and the APIs are not intended to be novice-friendly, it is this category of game-based education that is by far the most inspiring for young novices (as 5.3 demonstrates).
- *Coding as a core game mechanic*. Some games incorporate coding as a core part of the gameplay. Although this approach is relatively new, there are enough examples that we can identify at least two sub-genres (and there may, in fact, be more):
  - *Turtle games* – where the gameplay revolves around programming the movements of a "turtle" (which we've named after the programmable turtle from the 70s Logo environment [Pap80]). Examples are Lightbot [lig12] and the recent Code.org Hour of Code web games. In turtle games, the objective is usually to move the turtle from one point to another, avoiding obstacles in the process. In addition to turtle movement, some games allow other aspects of turtles to be programmed – such as the ability for turtles to shoot guns and/or to battle other players' turtles (e.g., Robocode [OG06]).
  - *First-person games* – where the player is embodied inside a game world and can interact with the world in a variety ways that have little or nothing to

do do with coding; but the player is *also* given the ability to write code that affects the world. Usually the coding mechanic is interwoven into the game's overarching metaphor. In CodeSpells [FEG13, EFG13c], for example, the player is a wizard who can run, jump, and pick things up – but who can also open a spell book and craft magic spells by writing code. In Hack 'n' Slash, the player has a special USB drive which can be plugged in to various game entities, allowing the player to manipulate their variables such as health and damage.

None of the above categories are mutually exclusive. For example, there is overlap between *turtle games* and *first-person games*. CodeSpells and Minecraft both allow turtle-like game entities to be programmed. The programmable turtle is a mechanic – which can either be employed as the game's sole mechanic (as in Lightbot), or as one of many other game mechanics (as in CodeSpells), or as a means for assisting modding (as in Minecraft). (We employed the turtle extensively in 5.5 and 5.7.)

Also, the *game-like IDE* is a feature that may appear in coding games or modding interfaces but doesn't have to. The original CodeSpells did not have a gamified IDE, instead requiring players to code spells in Java through an in-game text editor. The more recent remake of CodeSpells has switched to using a drag-and-drop language to help preserve the players' immersion in the game. Like the turtle mechanic, the gamified IDE is an option that may be included by the designer at will. When it comes to modding interfaces, we tried both a traditional IDE and a game-like one in 5.5.

Even the *building games* technique can coexist with the *first-person game* technique. The most recent version of CodeSpells allows for new mini-games to be created within the game itself.

The takeaway here is that the landscape of gamified coding experiences is a collection of systems that employ some set of the techniques we have outlined above. In light of the above, this chapter can be understood as follows:

- 5.3 shows that the *modding an existing game* technique is of high interest to young people between the ages of 10 and 15.
- 5.5 shows that the *modding an existing game* technique alone is sub-optimal but can be improved dramatically with a *gamified IDE*, *turtle mechanics*, and various tweaks to the classroom environment.
- 5.7 compares a Minecraft-based coding environment employing *turtle mechanics*, *gamified IDE*, and *modding an existing game* techniques with a Scratch-based environment employing *turtle mechanics* and *gamified IDE*. The point was to study how the *modding an existing game* technique affected students – independent of the other techniques.

### 5.2.1 Coding Games vs Other Games

Coding games remain significantly less popular than the average commercial and indie game. Although designers continue to seek for the holy grail of game-based coding education, there has not yet been a break-out success in the coding game genre. The Hour of Code turtle games attract millions of players – yet the very fact that there must exist a worldwide campaign to get kids to code for an hour is a telling indicator of how much more work there is to be done. Furthermore, there have been some successful Kickstarters for coding games: e.g., CodeHero (which raised over $150,000 but then failed permanently in development), CodeSpells (which raised over $100,000 and is currently being developed), CodeMancer (which raised over $50,000 and is being developed). This shows that at least the crowd funding community is excited about seeing these games developed. It remains to be seen, however, whether these games will achieve the kind of viral popularity of successful commercial (or even indie) games.

These games will need to prove that they can survive in a market populated by games that have simpler mechanics and lower learning curves. That said, the results of this chapter indicate that a sufficiently engaging non-coding gameplay experience does indeed inspire a desire to wrestle with the more difficult experience of coding – at least insofar as the game of Minecraft is concerned. This is encouraging and has important ramifications for games that seek to incorporate coding as one of their game mechanics.

### 5.2.2 Minecraft

#### The Game

Minecraft is usually described as an "open world" and a "sandbox" game. There is no storyline that the player follows. Instead the player (or players, if the mode is multiplayer) may explore a procedurally generated world that extends seemingly infinitely in all directions. The world is composed of trees, mountains, valleys, rivers, and oceans – but all of these things are themselves composed of 1-meter cubes. These cubes (called "blocks") come in various materials – e.g., wood, dirt, stone, diamond, etc. The player has the ability to break blocks, pick blocks up, and place them elsewhere. Players can create complex 3D structures out of these digital Legos. Players can also collect various materials and combine them together to make tools (e.g., a stone pickaxe which helps break blocks more efficiently) – a process called "crafting". The gameplay in Minecraft is fundamentally composed of just a few things: 1) exploration, 2) building with blocks, 3) destroying blocks to gather materials for crafting, and 4) crafting new items. When the game is played in "Survival Mode", players can die from various things (e.g., zombies, lava, falling, etc.). When the game is played in "Creative Mode" players cannot die; and it is also unnecessary to gather materials or craft because all of the game's items and blocks are made available to the player.

Minecraft was recently acquired by Microsoft for $2.5 billion. This is the largest video game acquisition in history. And the game itself is one of the most popular of all time. It is being used in education and research more and more frequently [Dun11, RSL12, SC13, WT13].

#### Modding

Because Minecraft is a multiplayer game, with a client and a server, Minecraft modding takes two forms: client-side modding and server-side modding. There is much overlap with regard to what can be accomplished in each kind of modding. A detailed examination of the differences is beyond the scope of this chapter.

The important thing to note about both client- and server-side modding is that the tools used are industry standard tools. For example, server-side modding involves writing Java code that interacts with an API that, in turn, interacts with the Minecraft server. Via the Java API, the game's state can be modified, and various "event hooks" allow the programmer to register callbacks that are triggered when, for example, a player moves or a block is broken. A standard way of writing a mod would be to download a jar file for a modding API – e.g., Bukkit or Forge – and to use an IDE like Eclipse to write code against this API. Once written, mods can then be deployed as a jar file to a Minecraft client or server. The experience is no more gamified than any coding project[1].

### 5.2.3 Evaluations Overview

In the course of our investigation, we ran three studies, one after another. Each study was designed (based on the results of the previous ones) to further our sustained investigation into game-based coding education. Before discussing each study in detail, we give an overview of the research questions that we addressed: What is the interest-

[1]At least this was the case prior to the software we designed in 5.5.

level in game-based coding in the 10- to 15-year-old age demographic? How attractive is a Minecraft-based coding educational environment? Can a Minecraft-based environment be an effective educatonal tool? What changes must be made to a Minecraft-based environment to optimize the educational value? And (after several weeks of optimization) how does a Minecraft-based environment compare to a more traditional Scratch-based environment?

## 5.3 Method

Instead of blindly assuming that the attractiveness of video games would make for an attractive educational offering, we tested this hypothesis by attempting to recruit students for 3 different kinds of courses. We designed the courses to incorporate games in 3 different ways, according to the taxonomy mentioned in the previous section. We advertised each of these courses over the same mailing list and social media channels at the same time of year. For each course, we solicited students between the ages of 10 and 15. Here are the course taglines as advertised:

- *Scratch Game Design Course*. Learn how to design and build 2D games with Scratch.
- *CodeSpells Java Course*. Learn Java by writing your own magic spells inside a 3D game.
- *Minecraft Modding Course*. Learn Java by learning how to mod Minecraft.

Each course was identically priced – costing $270 for 12 hours of in-class instruction, plus homework. Each course awarded college prep credit through a local university. Each course was to run on weekends for 8 weeks (1.5 hours per class). CodeSpells and

Minecraft were both advertised as Saturday courses; and the Scratch class was advertised as a Sunday course.

Courses were not free. We recognize that charging for admission makes our results only generalizeable to the socio-economic class in which parents can afford to purchase a $270 introduction to coding class for their children. We also recognize that further research would need to be done in order to parse out what role the parents played in the decision making process versus the children.

## 5.4 Results

The CodeSpells class received 3 signups (and was cancelled due to low enrollments). The Scratch class received 15 signups (2 female). And the Minecraft class received 87 signups (28 female).

This result was surprising: The interest in the Minecraft course was much higher than expected, and the interest in CodeSpells was much lower. The dichotomy was especially surprising given that both courses involved teaching Java using a 3D game and were being offered on the same day. The message is clear: It's not that the use of a 3D game makes educational opportunities inherently exciting. The particularities of the 3D game matter. We suspect that students are significantly more motivated to engage in educational opportunities that involve a game that they (and their friends) already play[2].

In light of the recent trends toward building educational games, our result constitutes a word of warning to game designers and researchers. No matter how well-designed an educational game is, there is an uphill battle to be fought against non-educational games that students already enjoy. Perhaps more effective is to reappropriate popular games for educational purposes, rather than embarking on the costly journey of designing

[2]A poll on the first day of the Minecraft class revealed that more than 90% of the students were regular Minecraft players.

a new educational game. This is the avenue we took in 5.5.

## 5.5 Method

With the question of attractiveness settled, we identified other key questions to address: Would a Minecraft-coding environment retain students in the long term? Would the game-centric course actually teach anything? Would the use of Minecraft in the classroom be logistically feasible?

We taught 3 iterations of the Minecraft-based coding course, making significant changes on each iteration. Our design goals were three-fold: 1) decrease the number of teaching assistants required to make the classroom manageable, 2) increase the amount of subject-matter covered throughout the course, and 3) allow for repeatability (i.e., other educators and researchers ought to be able to adopt the same methodologies and software tools).

We made some arbitrary decisions at the outset, which we describe in the next few subsections.

### Classroom Structure

One of the authors, an experienced K-12 classroom educator, was appointed to teach the first iteration of the course. The classroom we selected was a typical computer lab – with all workstations facing a central projector. We allowed 30 students to participate in the first iteration of the course. We decided to employ 5 teaching assistants.

### Curriculum

We wanted to design curriculum that would make coding education relevant to Minecraft players. So we interviewed several college-level Minecraft enthusiasts (all of whom had some modding experience) and identified two Minecraft-related activities that could become “hooks” for showcasing the power of coding for the average Minecraft player:

- *Building things*. Minecraft is fundamentally a game about constructing new things. Players build incredibly complex environments with blocks – such as castles, scale models of the Starship Enterprise, replicas of their home towns, and an entire continent from *Game of Thrones*. Because Minecraft only permits players to place one block at a time, there is great utility in being able to automate some or all of a construction project using code. To facilitate this, we designed some of our curriculum around programming a “turtle” to move in 3D space and build things.
- *Playing mini-games*. The unmodded version of Minecraft is a game without a clear objective. However, there exist many popular “mini-games” that can be played inside of the world of Minecraft by joining a “mini-game server” which adds new gameplay rules. Examples of mini-games are capture the flag, tag, and king of the hill. We decided to build curriculum around the design of mini-games inside of Minecraft.

Given the above, here are topics we wanted to cover in the course:

- *Minecraft’s command line*. The game of Minecraft ships with an in-game command line for doing basic game-state changes. We intended the use of the command line to be a basic introduction to syntax and semantics.
- *Scripting in-game commands*. Using a programming language, the in-game commands can be sequenced together, looped over, etc.
- *Turtle API*. The turtle’s API allows it to move in six directions: up, down, left, right, forward, and backward. It can also place blocks of any type found in Minecraft

(e.g., dirt, stone, obsidian, diamond, wool, water, lava, etc.)

- *Turtle+Loops*. Using the turtle API along with looping constructs immediately allows coders to build structures that would take days if built by hand – i.e., towers to the sky, massive cubes, stairways that go deep into the ground, etc.
- *Turtle+Functions*. It quickly becomes hard to reason about large projects without building new APIs on top of the turtle's low-level API. This module introduces making functions for common operations – e.g., building lines and squares.
- *Functions as Event callbacks*. After a function has been defined, it can be registered as an event callback, triggered automatically by the Minecraft server when game events occur – e.g., the player moves, the player breaks a block, an arrow lands, a monster attacks you, etc.
- *An example mini-game*. By registering an event that removes a block below a player whenever they move, the popular mini-game of Spleef can be coded. In this game, players run around an arena suspended over lava. The floor's blocks disappear a few seconds after a player walks over them. As the floor gradually vanishes, the last player to avoid falling to their death wins. This final module ties together the turtle (which can build the arena and remove blocks beneath the players) and the event system to make a mini-game that Minecraft players are familiar with.

## Programming Language

Between the prior study and this study, we decided against using Java. Although we had advertised the course as a Java course, we decided at the last minute to switch to JavaScript after discovering a Minecraft mod called ScriptCraft that allows JavaScript-based mods to be evaluated at runtime. We felt that JavaScript would allow for a tighter development cycle, eliminating the need to teach students how to compile Java code into Jar files, deploy them, restart the server, etc. Before the class started, we were able to build a web app where students could write code in an online JavaScript editor, press a button, and immediately see their code execute in Minecraft.

Students and parents either didn't notice or didn't care about the Java-to-JavaScript switch. No one mentioned it[3].

## 5.6 Results

- *Iteration 1.* On this 8-week iteration, we employed 5 teaching assistants. Yet we only covered material up to the *Turtle+Functions* module. Reasons for this poor outcome were the use of JavaScript and unexpected difficulties keeping students on task.

  - *Problem: JavaScript.* Half-way through iteration #1, we realized that JavaScript was a nightmare in terms of classroom management. The teaching assistants became full-time syntax error checkers. The dynamic nature of JavaScript meant that students would only realize that their code was wrong after they had written it and run it in Minecraft.

  - *Problem: Keeping Students on Task.* As it turned out, Minecraft is popular and addictive for a reason – and the classroom setting didn't change that. It was extremely difficult to get kids to focus on classroom activities, rather than simply playing Minecraft. While students were waiting for teaching assistants to come identify their syntax errors, they would discover "more important" tasks such as killing all the zombies in the area or building a house – rather than more productive tasks like trying to figure out why their code

[3]It's likely that most of them didn't know the difference between the two languages.

wasn't running. This caused us to decide that on the next iteration, we would not hold the class in a traditional computer lab, opting instead for a pod-based classroom setup where it would be easier for teaching assistants to monitor students.

  - *Problem: Server crashes.* With 30 students all running code on the same classroom server, crashes and restarts were frequent. During the few minutes in which the server was resetting, students could not test their code – which gave them yet another excuse to busy themselves inside of Minecraft (switching to single-player mode while the server was unavailable). Indeed, after an examination of the server's crash logs, we began to suspect that some students were attempting to crash the server on purpose (by writing infinite loops) – perhaps because disrupting the class was more entertaining than building things with the turtle. We decided to give students private servers in the next iteration.

- *Iteration 2* On this iteration, we increased the number of students we studied – forming 3 cohorts of about 26 students each. Each cohort met at a different time on Saturday. Instead of one of the authors teaching the courses, we appointed a lead instructor for each class (to try to ensure that the software and educational experience we were designing would be generalizable to other educators). We maintained the same number of teaching assistants. We switched to a classroom with a pod-based arragement (6 tables, each with 4 to 6 students facing each other). We required students to work in pairs, each sharing a computer and a server. We hoped the pair programming would help keep students on task. The most dramatic change was that we used Google's Blockly software to build a Scratch-like visual programming layer atop JavaScript – to cut down on syntax errors (see Figure 5.3).

– *Improvement: More Material Covered*. In this class, we were able to reach the middle of the *Functions as event callbacks* module. The reason for the improvement was that less time was wasted with class-wide server restarts and because Google's Blockly language made syntax errors impossible. Teaching assistants' debugging efforts were spent only on logic errors and type errors.

– *Problem: Students Complained about Working in Pairs*. Students were instructed to use pair programming, and teaching assistants were instructed to encourage this. We enforced a two-minute switching policy. However, we would frequently observe the non-coding student in the pair to be disengaged and bored. Having anecdotally observed this same phenomenon in our Scratch classes, we did not think (at the time) that this was a phenomenon linked to Minecraft. It wasn't until later, when we compared Scratch and Minecraft in the fineal study that we realized that Minecraft may actually make collaboration difficult for students. Hence, it did not occur to us to try to solve this problem for the next iteration.

– *Slight-improvement: Students Still Not on Task*. It was still a daily struggle to keep kids engaged with coding, rather than simply playing Minecraft. There was a slight improvement, however, in that students would test their code (e.g., to build a tower of TNT and/or to spawn 100 monsters) and then get distracted by non-pedagogical side effects (e.g., to blow up their tower of TNT and/or to kill all the monsters). Although these moments of off-task activity were not optimal, they were at least evidence that there was some seamless integration between coding and gameplay – i.e., coding activities created new opportunities for play. This was certainly better than students playing Minecraft because they were waiting for someone to come fix their syntax errors.

- *Iteration 3*. Near the end of iteration 2, one of the parents of a student in the class suggested that it would be nice if she could observe what the student was doing during class, and she was worried that he might be playing too much Minecraft during the class. This gave us the idea that we could track exactly how much code students were writing during class – a trivial feature to implement, since students wer coding in a web app that we had built, and it was easy to capture usage data. We also decided to use the projector to display a "leaderboard" during class – showing how many mods and lines of code each student had written that day. In other words, we gamified the act of writing code. This had a profound impact on students staying on task.

  - *Improvement: Students Staying on Task*. With the projected leaderboard, everyone's progress (or lack thereof) became public knowledge. Not only were students aware of whether they were falling behind, but the teaching assistants could see at a glance which students needed assistance. We were able to reduce the number of teaching assistants to 3 (and could probably have reduced it further if necessary). And we were able to quantify for ourselves and for the students how "on task" they were. We set a goal that each student should write at least 3 mods and 20 lines of code during each class. Students were consistently able to achieve this and more. As a result, we were able to complete all of the intended material – up to and including the final module on mini-game programming.

It is interesting to note that our use of gamification (the community leaderboard) was necessary in order to solve a problem that arose from game-based education itself: i.e., that Minecraft is irresistibly engaging. It's so engaging that students are willing to spend their Saturdays learning to code. But it's also so engaging that they can't focus on coding during class. It's a paradoxical state of affairs; but we were lucky to happen upon a gamification technique that mitigated the problem.

After 3 iterations we were able to answer our original questions in the affirmative:

- Would a Minecraft-coding environment retain students in the long term? Yes, the course drop-out rate was less than 1%.
- Would the use of Minecraft in the classroom be logistically feasible? Yes, but it was a struggle to design software and techniques to make it so.
- Would students learn anything in the game-centric course? In a database, we have stored every program that a student has written via our webapp, in addition to the version control history of each program. Although our future work involves performing a detailed analysis of these programs and their evolution, we can say definitively now that all students in Iteration #3 were able to write original code that used and combined the concepts covered taught in class: in-game Minecraft commands, the turtle API, loops, functions, and event callbacks. Below, we also give present ways that students used code to solve problems that were relevant to them as Minecraft players.

## Relevance of Coding

Always, a crucial learning goal is to make coding relevant to children. This is true regardless of whether the class uses Minecraft-based software. However, the use of Minecraft allowed us to observe two trends in which students incorporated the output of their coding into their gameplay.

- *Epic scale*. The first trend was the creation of things at a larger scale than would be feasible by hand. Students frequently used the word 'epic' to describe these creations. Figure 5.1 shows two examples: one in which many miles of in-game space have been covered in architecture created by code; and one in which an extremely rare in-game phenomenon (the super-charged Creeper) has been replicated many times in a student-build laboratory.

- *Power tool*. The second trend was a more sophisticated use of the building things at *epic scale*. It is the use of coding as a tool to accomplish a large-scale engineering project. In these projects, coding is used for what it is good for, whereas manual labor is also strategically employed to do the more artistic, detail-work. Figure 5.2 shows two examples: both of which involve massive architectural undertakings that were performed by a scripted turtle – along with artistic decorations added after the fact.

We were happy to see these phenomena because they indicated an understanding of the utility of coding: i.e., to automate manual processes. Minecraft is able to encourage this deep discovery because it is full of manual processes that students perform frequently.

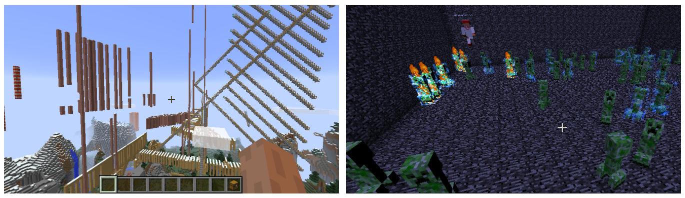


**Figure 5.1**: Two examples of students experimenting with things at ‘epic scale’. On the top, the student has stairways and archways in large numbers over the span of miles of in-game space. On the bottom, the student wrote code to create a holding container, then wrote code to spawn large numbers of Creepers, striking them with lightning during the spawning process. This caused the Creepers to become ‘super-charged’ – normally a rare occurrence in the games.

## 5.7 Method

Prior to this study, we had established two things: 1) the prospect of game-based (or at least Minecraft-based) coding education is an attractive prospect for students

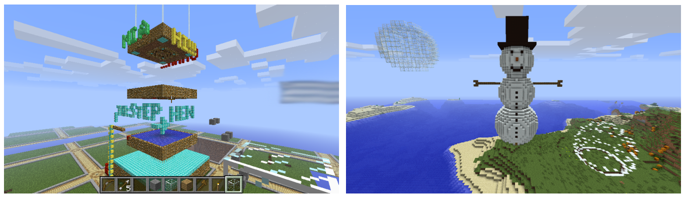


**Figure 5.2**: Examples of students experimenting with coding as a 'power tool' for building. The snowman's spheres and cube-shaped hat were generated with code. The details (buttons, face, and nose) were added by-hand afterward. Likewise, in the bottom picture, the large containers were generated by code, whereas the text decoration was crafted by hand.

between the ages of 10 and 15, and 2) Minecraft can be an effective tool in a game-based educational experience. This raised the question, How effective was it? To gain traction on this question, we wanted to compare the Minecraft-based experience to a more traditional coding education experience. We considered drawing comparisons between our Minecraft class and our Scratch class. However, this seemed unfair, given that the teachers, software, and curriculum were all different between the two classes. To reduce the number of variables, we opted for a more controlled lab study to address our research questions.

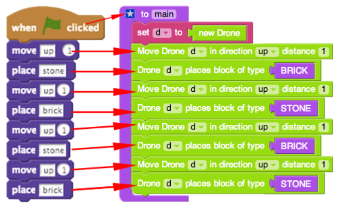
### Research Questions

- *Question 1*. Is a Minecraft-based coding experience more enjoyable than the more traditional Scratch alternative? And why?
- *Question 2*. Do students learn more in a Minecraft-based coding experience? .

### Study Design

We created two 3-hour-long "introduction to coding" workshops for each group: a Scratch-based workshop and a Minecraft-based workshop. We took care to make aspects of the two workshops as isomorphic as possible. For example, to control for differences in the usability of the Scratch environment versus the Minecraft modding environment, we used the Google Blockly version of the modding IDE. To control for differences in the usability of the Scratch API versus the Minecraft API, we extended Scratch with a turtle API. The only difference was that in the Scratch version, the turtle had only four degrees of freedom (up, down, left, and right) and could place 2D squares instead of 3D blocks. Using the same API and the same kind of graphical programming language allowed us to make the example code used in each workshop isomorphic (Compare Figures 5.3 and 5.4).

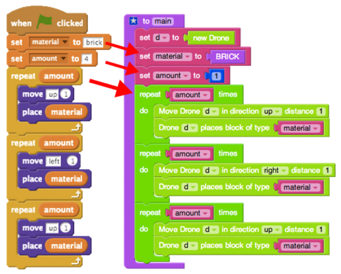
To control for the inherent intuitiveness of Minecraft's basic primitives (e.g., blocks of stone, dirt, grass, etc.), we designed 2D versions of these basic primitives for the Scratch environment, allowing for the same vocabulary to be used by the instructor at each of the workshops. For example, we could say, "Now, build a tower of stone blocks" at either workshop; and the phrasing would be meaningful in both contexts.



```
when clicked                to main
                              set d to new Drone
move up 1                     Move Drone d in direction up distance 1
place stone                   Drone d places block of type BRICK
move up 1                     Move Drone d in direction up distance 1
place brick                   Drone d places block of type STONE
move up 1                     Move Drone d in direction up distance 1
place stone                   Drone d places block of type BRICK
move up 1                     Move Drone d in direction up distance 1
place brick                   Drone d places block of type STONE
```

**Figure 5.3**: The first code snippet introduced in both the Scratch (left) and Minecraft (right) environments. Arrows demonstrate the program isomorphism we maintained between both environments.

This allowed us to develop one script to be used at each workshop. Each workshop was structured as a series of activities introduced by the instructor. The activities were explained using identical language (e.g., "You're going going to build a house with stone walls and a wooden roof. Here's an example of how to build a stone wall"). Then students were given time to experiment and play for the remainder of the activity. There were 5 such activities, the first and last of which are shown in figures 5.3 and 5.4 respectively.



```
when clicked
set material to brick
set amount to 4
repeat amount
    move up 1
    place material
repeat amount
    move left 1
    place material
repeat amount
    move up 1
    place material
```

```
to main
    set d to new Drone
    set material to BRICK
    set amount to 1
    repeat amount times
    do  Move Drone d in direction up distance 1
        Drone d places block of type material
    repeat amount times
    do  Move Drone d in direction right distance 1
        Drone d places block of type material
    repeat amount times
    do  Move Drone d in direction up distance 1
        Drone d places block of type material
```

**Figure 5.4**: A more complex code snippet introduced in the Scratch study, but which we did not have time to introduce in the Minecraft workshop. The greater complexity of the Minecraft environment led to less conceptual material being covered. Arrows demonstrate the program isomorphism we maintained between both environments.

The same instructor (one of the authors) taught both of the workshops – having extensive experience teaching coding with both Scratch and Minecraft.

We recruited 20 middle school students (ages 10 to 13), specifically calling for subjects with zero programming experience. Advertised payment for subjects was $10 per hour. We partitioned the subjects at random into two groups of 10. To one group, we administered the Scratch workshop, and to the other group, the Minecraft workshop.

In essence, what we ended up comparing were two almost-identical coding interfaces, one that involved placing cubes in the game of Minecraft, the other that involved placing squares on the Scratch canvas. What we were trying to isolate was the contribution of the "3D gameness" of Minecraft to the overall experience – i.e., the fact that the cubes placed in the game of Minecraft can be interacted with in the first-person, whereas the squares on the Scratch canvas were not interactive.

## Data Collection Methods

- Two of the activities included collecting student responses (to ensure that students were understanding the material). On these activities, the instructor asked students to draw a picture of what the code would produce when run. We graded these responses.

- In the middle of each workshop we gave students a snack break and told them they could do anything they wanted on the computer. We took notes on whether students elected to continue coding during this free time.
- We required each student to fill out an exit questionnaire.

## 5.8 Results

### 5.8.1 Question 1

*Is a Minecraft-based coding experience more enjoyable than the more traditional Scratch alternative? And why?*

The short (and surprising) answer is: the Minecraft-based workshop was rated as *less* enjoyable. This could be because it was actually less enjoyable, or it could be that students in different conditions interpreted the survey differently.

Our exit survey contained four 5-point Likert-scale responses ranging from 1 (“strongly disagree”) to 5 (“strongly agree”). The average results are tabulated below:

| | Scratch | Minecraft | |
|---|---|---|---|
| Today’s session was educational | 4.40 | 3.60 | p=.0093 |
| Today’s session was fun. | 4.60 | 4.05 | p=.0577 |
| I would like to continue doing what I saw in today’s session. | 3.90 | 3.60 | p=0.4 |
| I would recommend today’s session to a friend. | 4.30 | 4.0 | p=0.4 |

On all questions, the Scratch workshop participants rated the experience more highly. The rightmost column shows the results of a t-test, showing that the difference in the first question results are highly significant and that the second question results are on the threshold of being significant, whereas the final two question results are not significant. This could mean that students in the Minecraft condition really felt the session was less educational and less fun, or that their frame of reference was different. Doing something educational in the context of a something they usually do for fun (Minecraft) may cause them to interpret the experience as a non-educational one and one that feels less fun than their usual Minecraft play sessions. There is precident for this kind of thing [WKS98].

During the snack-break test, one pair of students in the Minecraft workshop was observed writing code, as was one pair of students in the Scratch workshop. The rest (in *both*) workshops were playing Minecraft[4]. It's hard to draw any conclusions from this except that playing Minecraft is a more compelling way to spend one's free time than coding in Minecraft – hardly a surprising result after 5.5 revealed the difficulties involved with preventing students from playing Minecraft during class.

### 5.8.2 Question 2

*Do students learn more in a Minecraft-based coding experience?*

We tested students' understanding by collecting written responses to questions posed by the instructor. Questions were of the form "Draw a picture of what this code will build". Students in both groups performed equally well on their responses to these questions (i.e., everyone answered the questions correctly).

However, the Scratch group exhibited two significant precursors to learning that indicate a more productive educational experience:

- *Faster coverage of material.* The Scratch group progressed much more quickly through the material – finishing the final activity (see Figure 5.4) by the 2-hour mark. The Minecraft group had only finished 4 of the 5 activities by the 3-hour

[4]This was an unexpected side-effect of using the same workstations in both workshops. Minecraft was installed and linked on the desktop; the Scratch participants found it and started playing.

mark. This means that the Scratch workshop was able to *cover material almost twice as quickly*. The reason for this is that turtle challenges in 2D are *significantly* less complex than turtle challenges in 3D. Building a 2D house out of squares is much simpler than building a 3D house out of cubes. Since the same teacher taught both workshops and had taught extensively with both Scratch and Minecraft before, we think the differences are due to the increased complexity of Minecraft.

- *Better collaboration*. An unexpected result on the survey was that three of the participants in the Minecraft group gave the same answer to the question "What if anything could be improved about today's experience?" They each gave variations on the response: "I would rather not work with a partner." These responses came from three participants, none of whom had been partners with each other. We had not observed any particular strife between partners – aside from the usual disengagement of the partner who wasn't controlling the keyboard and mouse. This difficulty of working in pairs was consistent with our observations during 5.5. And it was not a sentiment that the Scratch participants mentioned on their survey answers.

## 5.9 Discussion

The big conundrum here is clear: According to 5.3, a Minecraft-based experience is almost 4 times as sought after as a Scratch-based experience. Yet for all its magnetism, Minecraft (even with a Scratch-like IDE that we designed ourselves) performs worse than Scratch with regard to classroom management, student attention, student learning, and collaboration (see 5.7).

### 5.9.1 Collaboration

Researchers may find it surprising that collaboration seems to be inhibited by Minecraft – especially in light of the fact that watching people play video games is a major pastime for gamers [KSC$^+$12]. Youtube is rife with "Let's play Minecraft" videos – in which Youtubers stream their play sessions to their viewers. Indeed, Minecraft even comes with Twitch support built in – to more easily facilitate streaming play sessions to viewers. We too expected that students would enjoy watching their partner do Minecraft related activities – or at least that they would enjoy it more than watching their partner code in Scratch. This, however, was not the case.

Minecraft is a game about creating things; and we suspect that it may be unpleasant for players to allow someone else to adjust their creations. Indeed, the experience may be analogous to sharing a sketchpad and co-sketching a picture – occasionally this can be a fun party game, but it is impossible to keep one's personal artistic vision intact.

### 5.9.2 Hybrid Path

If we want to attract students to the field of computer science, then Minecraft is a wonder drug for getting them in the door. But if we want to optimize student learning and collaboration, then Minecraft has some serious drawbacks when compared to Scratch (a simpler alternative). In the Minecraft class students covered classroom material half as quickly, rated the class to be less enjoyable, and collaborated less.

One path forward is to continue using Minecraft but to make concessions: 1) allow students to work alone in spite of the research that suggests that pair programming is pedagogically positive [NWF$^+$03, Pre05], and 2) resign ourselves to a slower coverage pace as students wrestle with the inherently greater complexity of 3D spatial reasoning.

There is a hybrid path, however – to use both Scratch and Minecraft together to obtain the best of both worlds. That is, Minecraft can be used to attract students, whereas coding concepts can first be taught in the simpler world of Scratch before being applied to the 3D world of Minecraft. The fact that we have developed a matching API for Scratch in 5.7 can assist with this.

The simplification using Scratch is consistent with what Cognitive-Load Theory [Ovi06] might recommend: Learning is optimized when a learner's working memory is not overloaded. It is not unreasonable to hypothesize that the complexities of the 3D Minecraft world occupy more of one's working memory than the 2D Scratch canvas does, making the latter a better context to introduce new ideas. Things that may be in working memory in Minecraft are: the current time of day, the proximity of enemy creatures, the player's health level, roaming cattle or bigs, the efficacy the player's shelter to ward off monsters, etc.

In Minecraft, lightning can strike, creepers can blow up your house, zombies can attack you from behind, your roof can catch on fire, etc. These complexities are exactly what makes the game so rich and addictive. The world of Minecraft is filled with emergent experiences. Indeed, perhaps the desire to continually experience new Minecraft-related things is what makes students want to learn to code in Minecraft in the first place. Yet, at the same time, these complexities are like a fog that obscures the clarity of the concepts educators would like to convey.

Educational game designers and researchers should consider the trade-offs that our investigations into Minecraft-based education have identified:

- To make a game fun – designers should, by all means, try to follow in the footsteps of Minecraft. It is literally one of the most popular games in the history of gaming. Examples of things that appear to make Minecraft fun are: an open world with a non-linear story line, a focus on building and construction with an intuitive mechanic for doing so, and a tight integration between resource gathering, crafting of tools, and building.

- To make a system educational – complexity should be eschewed in favor of clarity and the reduction of cognitive load such that the user only ever has to focus on the material being learned. The fact that Minecraft is a complex world that presents the player with lots of things to do may not be beneficial in this regard.

Perhaps as the science of game-based education matures, we will discover design patterns for managing this fundamental trade-off.

## 5.10 Conclusion of Part 1

Game-based education continues to be of interest in the HCI community. We discovered that game-based education in the context of a blockbuster game had some surprising side-effects. Namely, that Minecraft-based education was almost 4 times as desirable – yet in the Minecraft class the class material was covered twice as slowly, even after several iterations of optimization and custom software development. Furthermore, for reasons that are not entirely clear, students in the Minecraft workshop collaborated less. Our results suggest interesting avenues for future research, as well as a hybrid path with regard to Minecraft-based coding education.

## 5.11 Part 2: Automated Tutoring + Blockbuster Game

From a simple premise, the entire field of educational games was born: for so many children, games are fun, but traditional learning is not. The same children who voluntarily spend hours playing video games have to be practically forced to do their homework. The goal of educational game design is to obtain the best of both worlds – an educational experience with the motivational power of a game.

Although the author of this thesis has been designing educational games for more than 10 years, this second half of the chapter presents an alternative to educational game design – one that we recently piloted to great effect. Our approach was to take an already-popular, commercial game and to design educational software on top of it. More precisely, we constructed an automated tutoring system (ATS) on top of a commercial game. We call such a system a G(ATS) (game plus automated tutoring system). The results presented in this chapter demonstrate that the G(ATS) we built encourages extremely high student motivation. This leads us to believe that G(ATS) design may be a promising new approach to building highly-motivating educational software.

The combination of an ATS with a commercial game is, as far as we know, a novel one. However, the ingredients of this combination are not new. There exists a wealth of research on automated tutoring systems [KP05b, AKR$^+$02, SG11] and on educational games [YZ09, LKLC11, HMA13, HMAM14]. There even exists a rich middle-ground – where gamification techniques or games have been combined with automated tutoring system techniques [SA98, GMGMGC04b, JDM10, GMT14, Eag09, EB12, MWM12b]. However, a G(ATS) should not be confused with an ATS that leverages gamification techniques (e.g., badges, simulations, progress bars, avatars, storylines, etc.), but rather the combination of an existing, already-popular game with an ATS. Indeed, a G(ATS) is better described as the ATSification of a game, rather than the gamification of an ATS.

Note that although we have never seen an ATS built on top of an existing commercial game, it is by no means new to use commercial games in educational contexts. Teachers have been incorporating commercial games in their classrooms for years. Indeed, the approach has been so successful that there now exist companies (e.g., TeacherGaming, LLC) that help teachers integrate games into curricula across a range of academic subjects. Inspired by teachers' intuitions about the motivational power of blockbuster games, the G(ATS) approach is to integrate a popular game into tutoring software. The goal, though, is the same: to motivate students and to present educational material in a context that children already care about.

Closed-source commercial games are rarely designed for software extensibility, however, which is likely why there have been so few G(ATS)s to date. With little (if any) published empirical data, there are two plausible viewpoints about the G(ATS) approach. The pessimistic viewpoint is that the combination will be perceived as what the research community has been calling "chocolate-covered broccoli" for over a decade now [FEG13, LKLC11, Bru99] – less educational than a traditional ATS and less fun than the pure game. The optimistic viewpoint is that the inclusion of an already-popular, non-educational game may incentivize deep and/or long-term engagement with the G(ATS).

To shed light on the issue, we wanted to answer various questions: Does the G(ATS) integrate well with classrooms? Does it motivate students? And is it educational? To answer these questions, we studied 71 students using LearnToMod, a G(ATS) for learning to code in the context of modding Minecraft. We used automatic data gathering techniques, direct observation, video analysis, and post-test data to build a picture of how students interacted with the G(ATS) at three different scales:

- Over a 6-week course. We automatically recorded interactions every hour for 6 weeks, discovering considerable system usage occurs outside of class. We administered a post-test at the end of the 6 weeks.

- During a 2-hour class. We automatically recorded interactions every minute for several 2-hour classes, discovering a cyclical pattern of on-task and off-task behaviour.

- Within off-task lapses. Using video data and direct observation, we characterized how students spend their time off-task – i.e., playing Minecraft.

On the negative side, 45% of time in class is spent off-task (much higher than prior research on off-task behaviour would suggest [BCKW04]). On the positive side, over 79% of students engaged in self-motivated activities with the system outside of class and all students scored above 80% on the post-test. We conclude this chapter with a discussion of how these results can inform G(ATS) design in general. Our hope is that more such systems will be built and studied. The early results in this space are quite promising.

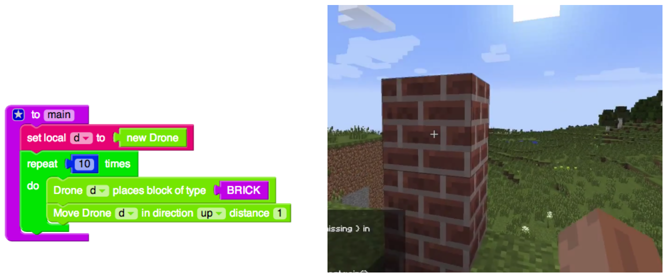
## 5.12 Background

The game of Minecraft is often described as "digital legos" because the hallmark of its gameplay is the construction and destruction of 1-meter 3D voxel blocks. Players find themselves in a voxel-based world where land formations like mountains, rivers, and trees are all constructed from voxels. Players can destroy these formations and/or dig into them to collect resources to build their own formations.

Although the game is not built to be "modded", an unofficial modding community has existed since the game's inception. Youtube videos featuring famous mods or modders routinely receive millions of views, demonstrating that modding is part of the "culture" of the game. Modding may involve adding new content to the game (e.g., new voxel types or new creatures) or extending the game with new rules (e.g., adding rules to implement capture the flag).

At the time of our study, LearnToMod was web-based software that was intended to be run alongside Minecraft.[5] It guides the user through a sequence of lessons, videos,

[5]It has since become more integrated with the game, capable of (optionally) being embedded into Minecraft.



```
to main
  set local d to new Drone
  repeat 10 times
  do  Drone d places block of type BRICK
      Move Drone d in direction up distance 1
```

**Figure 5.5**: A simple mod that builds a brick tower inside of Minecraft. More advanced structures can be built procedurally. APIs also exist to add new content and/or rules to the game. The bottom screenshot shows the top of a the brick tower built from the code. It is a physical structure that the user can interact with: e.g., break, climb on, build on to, etc.

and puzzles that involve writing Minecraft mods. These mods are written within LearnToMod's embedded IDE. An example of a program that might be written in the IDE or in a lesson is show in figure 5.5.

Although the software is designed to be usable in isolation, that is not its only usecase. It is also designed to be used by teachers in classrooms and has various classroom management and monitoring features for teachers. The goal is to allow teachers with little coding experience to run classes that teach coding in the context of modding Minecraft.

## 5.13 Methodology

### 5.13.1 Research Questions

Our main goal was to shed light on whether the G(ATS) approach was viable and generalizable. It is worth mentioning that we already knew the G(ATS) was popular; at the time of our study, LearnToMod was already being used in about 200 classrooms and by about 6000 users. Market validation can be fickle, though. So we had three research questions that we wanted to investigate empirically:

- *Does the G(ATS) integrate well in a classroom?* (Discussed in 5.14.2)

- *Does the G(ATS) motivate students?* (Discussed in 5.14.1)
- *Do students learn from the G(ATS)?* (Discussed in 5.14.3)

To answer these questions, we chose to do an in-the-wild study on LearnToMod. As is the case with many ATSs [WCBn$^+$11, BCKW04, Pil03, Soh06], a critical usecase is the deployment of the system within a classroom setting, where a teacher employs it alongside other educational tools (e.g., textbooks, blackboards, etc.). In this context, the ATS can provide adaptive tutoring while gathering detailed learning metrics, and the teacher can provide targeted instruction as necessary.

In the class we selected, LearnToMod was integrated as follows:

- LearnToMod was the primary educational tool. There was no textbook; and the teacher did less than 15 minutes of direct instruction. There was no homework.
- During class, LearnToMod was open in the web browser for students to use the ATS. They also had access to LearnToMod at home.
- During class, Minecraft was also running at all times on each student's computer. After writing code in LearnToMod, students could deploy it directly to Minecraft for execution.

The classroom site we chose holds four 2-hour classes every week. Each class is part of a 6-week course for middle school students (10 to 12). During the 6 weeks of our study, there were 71 students enrolled. The ATS gathered usage data for all of them during this timeframe. Of the four classes, we selected the one with the most students for more detailed observation: minute-by-minute automatic data gathering, direct observation, etc. This was the 10am-12pm class on Saturdays, which had 19 students enrolled.

*Long-term, hour-by-hour data*. We instrumented LearnToMod with the ability to gather hour-by-hour data for a population of users over a 6-week period. This was to assess whether students were using the AST outside of class. Because we selected a class that did not assign homework, a high frequency of at-home use would suggest self-motivation. The main data we collected at this 6-week scale was *program creation*. Any time students created a new program in the ATS's embedded IDE, this event's details were logged. Programs can be created by users as the ATS guides them through lessons, or they can be created by users who are engaging in "free-play" – coding without the direct guidance of the ATS. We recorded both types, along with the timestamp of the creation and the user who created them. Because the ATS was available outside of class, we were also able to detect program creations that occurred outside of the normal class hours.

*Short-term, minute-by-minute data (in class).* We also instrumented LearnToMod to gather minute-by-minute data while classes were in session. This was to quantify how much the software was being used in class. The main data we collected at this in-class scale was *program edits*. Any time students made a change to a program, we logged the timestamp, editing user, and type of change. Types of changes include: adding a line, deleting a line, changing a line, or moving a line. When a user is editing a program, many editing events can occur in the span of a few minutes. We used this data to obtain a quantitative measure of the 10am-12pm class's productivity from minute to minute during class.

We also set up cameras to gather footage of students interacting with the system. This allowed us to monitor activity while students were engaging with Minecraft and not with the AST. Because LearnToMod is a G(ATS), students were expected and encouraged to interact with the game as well as with the ATS – cyclically writing code in the ATS and then running it inside of Minecraft. Cameras were specifically placed so that students' computer screens were visible. Ten hours of in-class video of 20 students were then analysed by one of the authors. Analysis involved, for each student visible, identifying the major activity that the student was engaged in during each 30-second segment of footage.

Although there are many ways to categorize video data, we chose to investigate on-task versus off-task behaviour. The amount of time students spend off-task is a good indicator of how well the ATS integrates with the classroom environment, as well as how motivational it is for students. The classifications were as follows:

- *On-task*: Interacting with the ATS; talking with the teacher; or testing the effect of one's code in Minecraft.
- *Off-task*: Chatting with classmates; digging, building, or aimlessly traveling in the Minecraft world; or "unproductive code execution" – i.e., running code in unproductive ways.

When more than one "major activity" occurred during a 30-second period, all such activities were tallied.

To the untrained eye, it can be impossible to tell the difference between off-task and on-task behaviour inside of Minecraft. The analyst (one of the authors) had a year of teaching experience with LearnToMod and Minecraft and can be considered to be an expert in identifying productive in-Minecraft student behaviour.

*Post test*. We administered a 9-question post-test to 30 students (all students in the 10am-12pm class and a random sampling of students from other classes). Three questions on the exam tested content from the first day of the course (6 weeks prior); three questions tested content from the middle of the course (3 weeks prior); and three questions tested content from the end of the course (1 week prior). At the teacher's request, we designed the exam to be completed in less than 10 minutes, so as to avoid cutting into the scheduled activities for the final class day.

## 5.14 Results

At the high level, we found three things: 1) an encouraging number of students (79% of the 71 subjects) used the software at least once outside of class, 2) an alarming amount of class time (an average of 45% per student) was spent off-task playing Minecraft, and 3) all students scored above 80% on the post-test.

### 5.14.1 Use outside of class

The heatmap in figure 5.6 shows the use of the ATS during and outside of class. Each cell corresponds to an hour of time during a 6-week timeframe. Each row is an hour of the day; and each column is a day during the timeframe. Dark cells show heavy use (many programs created). Classes were in session at 10am-12pm on Saturdays, 12:30 to 2:30 on Saturdays, 3:00 to 5:00 on Saturdays, and 4:30 to 6:30 on Tuesdays. Note that the second Saturday and Tuesday in February were holidays, which is why the usage pattern is noticeably lighter on those days.

Usage tends to begin after 6am, with heavier use in the evenings. On weekdays, usage is sparse until about 1pm (when some schools in the area begin to let out). Usage in the evenings is heavy between 4pm and 8pm and starts to decrease after that.

Furthermore, we determined that this observed usage pattern is not simply because a small group of eager students is using the software frequently. As mentioned previously, 79% of students used the software at least once outside of class. More specifically, figure 5.7 gives an analysis of how many students were active outside of the day of their class. Each distinct color represents a distinct user. During hours where there were 1, 2, or 3 active users, we placed each of those users' colors into that hour's square. Black squares indicate more than 4 active users.

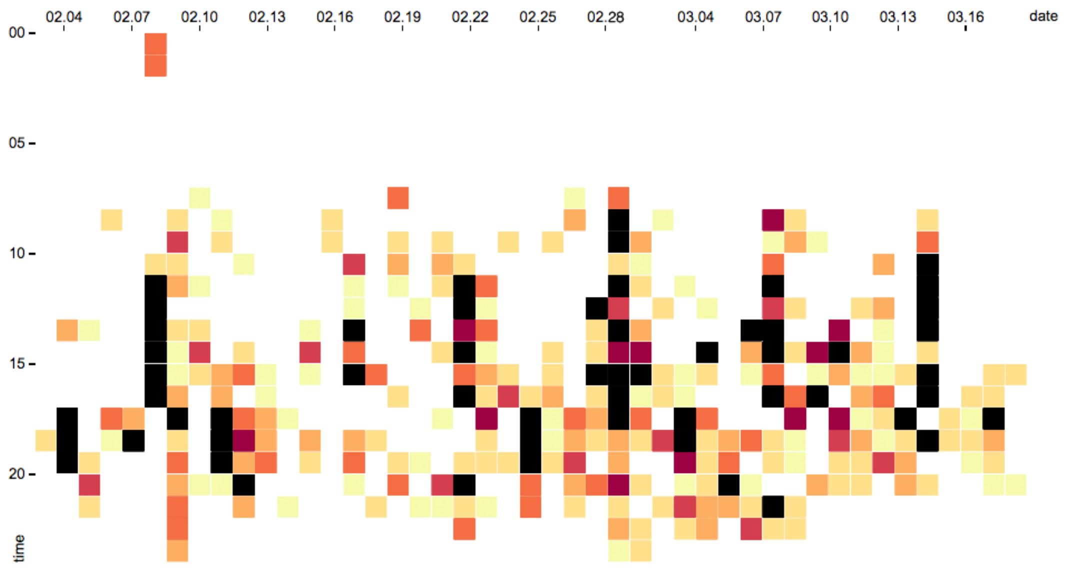


**Figure 5.6**: A 6-week heatmap, showing hour-by-hour system usage based on number of programs created in each hour. Black indicates more than 25 programs per hour. The light yellow indicates between 1 and 5 programs. Black columns show in-class use on each Saturday (except for the second Saturday in February, which was a holiday). The Tuesday evening class can also be clearly seen. 71 subjects' data are displayed here.

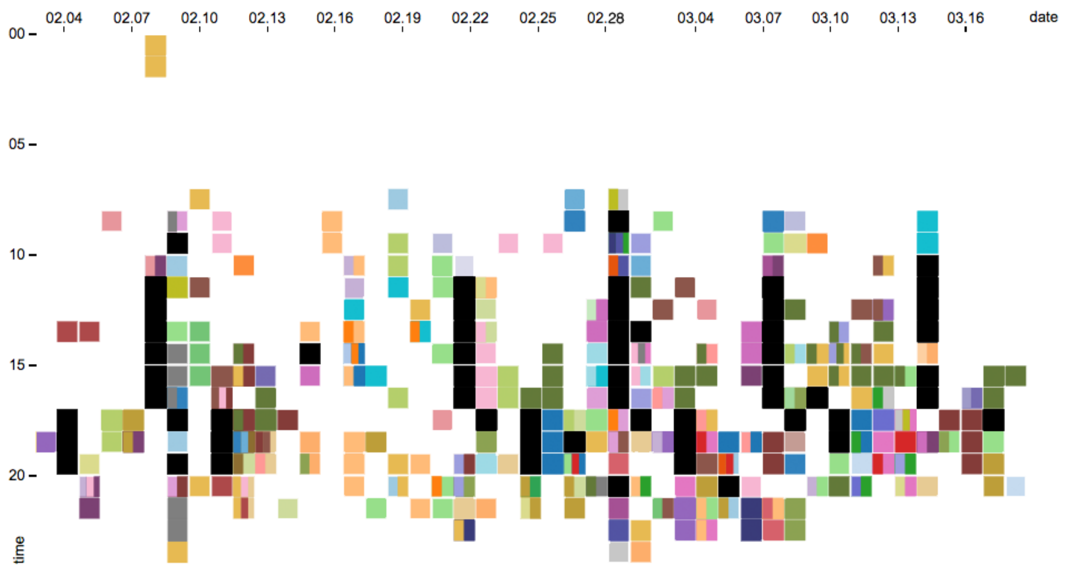


**Figure 5.7**: Distinct colors indicate distinct users. Squares with 1, 2, or 3 colors indicate that 1, 2, or 3 users created a program during that hour. Black squares indicate more than 3 active users during that hour. This proves that the usage data from figure 5.6 was not generated by a small subset of the subjects. There are more than 56 distinct colors here – about 79% of the 71 subjects.

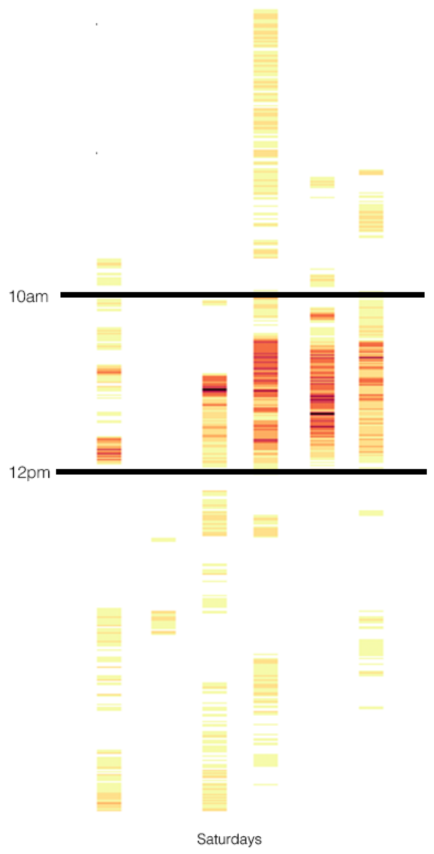


**Figure 5.8**: Six Saturdays are shown here (the second of which was a holiday). Minute-by-minute data for all 19 students enrolled in the 10am Saturday class are shown. Unlike figure 5.6, this data shows program edits, rather than program creations. White areas during the 10am to 12pm timeframe denote times when no students were editing code. These were lecture periods or group discussions. Dark areas indicate 85 or more edits per minute. Frequent use occurs both before and after classes.

### 5.14.2 Off-task behaviour during class

Figure 5.8 is a minute-by-minute heatmap that zooms in only on Saturdays and only on the 19 students enrolled in the 10am class. This heatmap differs from the hour-by-hour data of the coarse-grained heatmap in that it visualizes program edits instead of program creations. Students tend to edit programs more frequently than they create them, so this gives a more accurate minute-by-minute picture. In most cases, the minute-by-minute program edit heatmap aligns straightforwardly with the hour-by-hour program creation heatmap. However, there are some minor differences, like for example on the first 10am Saturday class, where the hour-by-hour heatmap shows heavy use, whereas the minute-by-minute heatmap shows lighter use. This indicates that the usage pattern on the first day of class was dominated by program creations rather than program edits. (We can confirm this because we directly observed the class and know that the students created many small programs, many of which were not edited.)

One interesting side-note: Although this data is intended to show usage patterns during class, it also further corroborates the aforementioned finding that usage occurs outside of class – in the hours both before and after.

In class, the usage rises and falls dramatically and sometimes cyclically, with the class-wide number of edits varying from 1 to 14 edits per minute to around 85 edits per minute. We know from direct observation that periods in where there is zero usage are times in which the class was engaged in a teacher-led group discussion or in which the teacher was demonstrating something on the projector.

We used video analysis to zoom in even further, identifying and examining moments when students were not engaging with the ATS but with the game of Minecraft. In some cases, this kind of interaction is expected and encouraged, and not classified as off-task. The G(ATS) and the class itself are built with the assumption that students will write code in the LearnToMod IDE and execute that code in Minecraft. Because the ATS guides students through lessons that deal with “interesting” code, the effect of the executing code is often meant to be appreciated and interacted with inside the game (e.g., creating massive structures within the game, or summoning huge numbers of Minecraft creatures, or registering new event callbacks that strike the player with lightning every time they dig a hole). So when a student is inside Minecraft climbing on structures or digging holes, this is not necessarily off-task. Even a moderate amount of admiration or interaction after running the code is expected, encouraged, and classified as on-task.

Activities that were *definitely* off-task, however, were the classic Minecraft activities that had nothing to do with the code that a student had written – i.e., digging, building, flying around, chatting with neighbors about Minecraft (not about coding), etc. All together, these activities comprised about half of the total time off task. This can be seen in the pie chart of the introductory figure 4.1.

The rest of the off-task time was classified as "unproductive code execution" because the students' activities did nominally pertain to the code that he or she had written, but which seemed to have no apparent pedagogical value. This is admittedly subjective and likely to vary if the experiment were repeated with a different analyst. That said, the analyst had over a year of teaching experience with LearnToMod and is a staunch constructivist who places great import on students playing with their software artifacts in various ways. When in doubt, the analyst assumed the student was on-task and tended toward off-task classifications only when the student's behaviour seemed to be particularly unproductive. Some examples from the analyst's notes: "student has executed the same Build a Tower mod over 100 times; the 100 towers are interesting but do not appear to be more interesting or informative than when there were 50 towers a minute ago." and "The student is running in-game commands that were taught on day 1, but this is the last week of class, and he has been consistently ignoring the teacher's instructions."

The off-task time totalled to 45% (SD=20%) of the class time. We discuss this in depth in our discussion section – e.g., whether this is a bad thing, what could be done about it, and how future studies can use this chapter's results to obtain a more empirical definition of "off-task" in gray areas like this one.

Also noteworthy (though not necessarily surprising) is that students spent the majority of their time (80% on average) in Minecraft – sometimes on-task and sometimes off-task.

Aside from time spent in Minecraft, we did not observe *any* other kind of "solitary off-task time", which Baker et al defines as: "any behavior that did not involve the tutoring software or another individual (such as reading a magazine or surfing the web)" [BCKW04]. Baker observed this kind of behaviour to take up about 78% of a student's off-task time – making the fact that we observed none of those behaviours significant. In other words, Minecraft dominated students' solitary off-task behaviour, making things like internet surfing or magazine reading conspicuously absent.

### 5.14.3 Post-test results

We administered a post-test because we had originally hoped to correlate system usage patterns and/or off-task behaviour with test performance. But there are two confounds here: 1) the teacher did not permit us to administer a pretest, citing the stress of first-day logistics, 2) the exam scores turned out to be higher than expected, without enough variance to show meaningful correlations.

However, the simple fact that all students scored higher than 80% on the post-test provides important insight into the previous results. Even students who had never taken a coding class before, and who would not have been expected to perform well on a pre-test, all scored above 80%. A low average score on this test would have cast some doubt on the benefit of spending time using the ATS outside of class or suggested that off-task behaviour during class is clearly detrimental. As it stands, the negative effects (if any) of in-class off-task behaviour did not appear to be reflected in the post-test scores. Whether this is due to the positive effects of frequent out-of-class use cannot be concluded without further investigation. The 80% score also provides a benchmark for future A/B studies that may seek to parse out the relative effectiveness of different parts of the G(ATS) ecosystem (the ATS, the game, the teacher, the in-class behaviour, the out-of-class behaviour, etc.)

## 5.15 Discussion

### 5.15.1 The G(ATS) Double-Edged Sword

Of key interest to us is the dichotomy that appears to arise directly from the fusion of an ATS and an extremely popular game. The instructor of the class we studied summed up the dichotomy succinctly: “Minecraft is like a black hole with a massive gravitational pull. It sucks students in. It compels them to take the course, but it also distracts them while they take it. It’s a double-edged sword.” When we asked the teacher about the 45% off-taskness, he said, “That’s a relief. I thought it would be even worse! I feel like every time I turn around, I have to tell a child to stop digging holes. It’s like they don’t even realize when they’ve been hypnotized into wasting extraordinary amounts of time.”

This chapter’s data corroborates the instructor’s insights. Students are excited about the class because it involves Minecraft.[6] Not surprisingly, during the class, the majority of a student’s time is spent in Minecraft. Although sometimes students are on-task in-game, the game does compel frequent off-task behaviour. That’s the double-edge: the game is extremely compelling – which makes it distract from the very educational experience that it incentivized in the first place.

If we had gathered only in-class data, we might have concluded from the high game use and high off-task behaviour that students have to be practically forced to engage with the educational material. However, the out-of-class usage patterns negate this conclusion. Even though students are free to do whatever they want outside of class, 79% of them *still* elected to engage in educational activities through the ATS. Many do so the moment school lets out. Many wake up early on weekends to do so. This is encouraging because it seems to suggest that the popularity of the game rubs off onto the

[6]Recruitment numbers are high: the teacher has taught more than 400 students in the past year.

ATS, resulting in "on-task" behaviour even when there is no "task" per se. The term for this is "intrinsic motivation" [RD00] – a desirable quality that educational software and teachers alike strive to nurture in students. Interestingly, there is precedent for techniques designed to increase motivation (e.g., "gamification") to backfire by *reducing* intrinsic motivation (see [DKR99] for an overview). So it is encouraging to see that the G(ATS) approach performs well with regard to this metric.

If we had solely gathered usage data and not administered a post-test, then we might have been left wondering whether the off-task in-class behaviour was harmful and whether the at-home usage was actually productive. Although the overall excellent scores on the post-test do not conclusively prove that off-task in-class behaviour is harmless nor that at-home usage is beneficial, it does seem to suggest that something about the overall G(ATS) ecosystem is functioning well.

### 5.15.2 Off-task behaviour

The high post-test scores coupled with high off-task behaviour leave us with an unanswered question: is off-task behaviour detrimental? The relatively well-accepted Caroll Hypothesis [Car89] says "yes", at least with regard to traditional classrooms. The Caroll Hypothesis has also been confirmed in the context of at least one ATS [RRB12]. So there are two possibilities with regard to our G(ATS): 1) Either the G(ATS) we studied somehow compensates for the detrimental effects of off-task time, or 2) our post-test was not sufficient to detect the detrimental effects.

*Possibility 1: The G(ATS) is special.* It is possible that the frequent out-of-class use of the software had a positive effect that offset the negative effects of off-task behaviour during class. In other words, the use of a popular game may have simultaneously increased off-task behaviour along with overall usage, with one effect canceling out the other.

*Possibility 2: Our post-test was not detailed enough.* Although the post-test does show that all students performed roughly equally, there are many learning metrics that we did not test. It is likely that future studies may find some correlation between off-task behaviour and various learning metrics.

These possibilities are not mutually exclusive and our conjecture is that both are in operation. We do think it is likely that out-of-class use had a beneficial effect. And we know from prior work on off-task behaviour that it is usually detrimental. We would be surprised if the beneficial effects canceled out the detrimental ones in a straightforward way. A more nuanced post-test is called for.

With that said, even if one were to conclude (based on prior research) that the observed off-task behaviour is suboptimal, we would still caution against naively trying to eliminate it. There is a delicate balance at work here: the very "gravitational pull" that pulls students off-task is also what drew them to the class in the first place and is plausibly what motivates them to use the ATS outside of class. To restrict a student's off-task time may require restricting their access to the game in such away that overall reduces their enjoyment of the G(ATS). This may reduce their time spent outside of class and may have an overall negative effect on their post-test scores. Anecdotally, we once taught a class with LearnToMod in which we had disabled the students' ability to move, dig, or build within the Minecraft world. The students were so visibly upset by our hobbling of the game that we ultimately reversed our decision and allowed unfettered access to the game.

Our point is that there are complex design decisions that arise when building or teaching with a G(ATS). The game and the ATS exist in a nuanced relationship. Care should be taken to productively channel a student's enthusiasm for the game into the ATS. There is much room for empirical research on the effectiveness of interventions that promote on-task behaviour. Furthermore, our results leave it an open question as to whether off-task behaviour is detrimental enough to warrant intervention in the first place.

### 5.15.3 “Unproductive code execution”

A gray area in the data is the non-trivial fraction of time labeled “unproductive code execution”. During this time, students were observed to be interacting with code they had written (i.e., executing it), but in ways that the analyst did not deem to be pedagogically fruitful. We ultimately labeled this behaviour as “off-task” because it did not appear to be an optimal use of class time.

For comparison, previously reported off-task behaviour ran at 20% for traditional classroom instruction in two prior studies [LKN99, LL86]. It was 18% in another study, which involved an ATS [BCKW04]. Another study showed off-task behaviour for ADHD students to be about 25%, compared to 12% for their peers [KRMA08]. Percentages of off-task behaviour in this ballpark could only be observed in our data if *all* of the analyst’s “unproductive code execution” classifications were considered to be on-task – a very forgiving analysis indeed. So all subjectivity aside, the amount of off-task behaviour observed appears to be quite high – perhaps more than double the expected amount.

Still, there is ambiguity, and we hope that future work will yield an objective, empirical evaluation of whether this phenomenon correlates with measurably poorer learning gains. Given the frequency of which students engaged in this behaviour, it is probably important to determine if it is good or bad.

We see this as a broad and critical question for research in the G(ATS) field: Which interactions with the game are productive, and which ones are not? And what interventions (automated or otherwise) can be employed to replace bad interactions with good interactions?

## 5.16 Beyond Minecraft and Coding: G(ATS) Design

The G(AST) concept is generalizable beyond the game of Minecraft and beyond the subject of coding. The key ingredient is a game (preferably one that students already love) that can be used as a context for education. With this in mind, it is easy to see how Minecraft and coding made for a strong pairing. Minecraft is the ultimate "sandbox" game – with open-ended gameplay and a preexisting culture of creative construction and modding. Coding is the ultimate abstract skill, which can operate on domains from robotics, to text processing, to animations, to Minecraft modding. Not only was their pairing a natural one, the game of Minecraft and the subject of coding both yield other G(ATS)s that are fairly easy to imagine.

*Coding G(ATS)s.* Many blockbuster games have thriving modding communities – e.g., Skyrim and Gary's Mod. This means that 1) the technical challenges of integrating an ATS into the game have already been solved, and 2) the culture of these games already revolves around programming. Building an ATS around one of these games would be reasonably straightforward. LearnToMod could be used as an exemplar – with Minecraft simply swapped out for a different game.

*Minecraft G(ATS)s.* Minecraft is already used in thousands of schools world-wide to help teach virtually every academic subject (see minecraftedu.com). Its open-ended, sandbox genre makes it a ready-made platform for education. Plus, its moddability means that the technical challenges of integrating an ATS into the game have been solved. One recipe for non-coding related G(ATS)s that involve Minecraft would be to observe how Minecraft is being used in, say, a math or history class and to ATSify the class.

*Other G(ATS)s.* Minecraft also gives insight into what sorts of other games might be ripe for ATSification. A productive litmus test might be: What popular games are being used in classrooms? Use of the games in classrooms can inspire and inform the construction of ATSs around them. Games like Kerbals Space Program are being used to teach physics and engineering (see kerbaledu.com). The game of Civilization has been used to teach and history, geography, economics, and politics [SB04]. What these games have in common is that they are not designed to be educational games, yet their gameplay (combined with their popularity) has catalyzed their adoption into educational contexts. Those contexts can be further formalized with an ATS.

### 5.16.1 Educational game design versus G(ATS) design

The critical difference in the philosophy of the G(ATS) designer – as opposed to the educational game designer – is that the G(ATS) designer looks for an already-popular game and seeks educational opportunities within that space. On the other hand, the educational game designer must build a game from scratch and, thus, is faced with the twin burdens of 1) building a fun game, and 2) making it educational. The recent trend is also to avoid "chocolate-covered broccoli" [FEG13, LKLC11, Bru99] by seamlessly interweaving the gameplay with the educational content. These are heavy burdens indeed. Even after the chocolate-free educational game is built, there is no guarantee that users will adopt it – even if it is excellent.

The G(ATS) approach sidesteps some of these issues but has unique challenges of its own. For one thing, an existing game can be just as difficult to build an ATS around as a new game is to build from scratch (though this technical burden is much lighter when dealing with games that already have thriving modding communities, solid APIs, and good online resources). Furthermore, as our analysis of LearnToMod has shown, the high engagement and adoption comes at a cost: the game portion of the G(ATS) is more fun to use than the ATS portion, resulting in higher than normal off-task behaviour.

Finally, the educational game designer has the liberty to pair any subject matter with any game type. A first-person sci-fi shooter with a funky retro vibe for teaching math is as plausible as a fantasy role-playing game with a film-noire storyline for teaching German. The G(ATS) designer, on the other hand, is restricted by what popular games exist at the moment, how easily they lend themselves to contextulizing a particular subject matter, and what APIs and software tools exist to interface with the game.

As long-time game designers, we feel that both approaches to designing educational software have their respective merits. G(ATS) design is a promising new direction, though, with unique challenges and opportunities. The empirical results support this.

## 5.17 Conclusion

This chapter identifies a new and promising genre of educational software – an ATS built "around" a popular video game. A G(ATS) is best described as the ATSification of a game, rather than the gamification of an ATS. We investigated whether the system integrated well with classrooms, as well as whether it was motivational and/or educational.

On the positive side, our results show that the G(ATS) was used frequently outside of class, suggesting that the combination of a popular game with an ATS motivates the use of the ATS during free time. On the negative side, the amount of off-task behaviour during class was high. Ultimately though, the post-test showed that students were retaining concepts taught throughout the course, suggesting that the off-task behaviour doesn't lead to poor learning outcomes.

These results lead us to advocate that the G(ATS) approach should be incorporated into the toolbox of educational software designers. Although we know from experience that such systems are non-trivial to build, we hope to see more such systems in the future. To that end, we have outlined how the G(ATS) approach may be applied more broadly than Minecraft and coding. With many popular moddable games on the market (e.g., Gary’s Mod, Skyrim, etc.), the G(ATS) design space is constantly growing.

## 5.18 Acknowledgements

This chapter uses work that was previously submitted for publication: Stephen R. Foster, Sorin Lerner, William G. Griswold. ”G(ATS): Games plus automated tutoring systems”

# Chapter 6

# Design Take-aways and Conclusions

When looked at individually, each of our three systems contributes to the body of knowledge within its distinct sub-genre. However, when looked at as a whole, the three systems together help us articulate a design space for educational coding games. It is this later, more holistic contribution that we believe will be more impactful in the long run. This chapter begins with an overview of each system and its contributions to knowledge. It concludes with an in-depth articulation of the trade-offs and features of the entire coding games design space, as informed by our experiences building and studying three distinct systems within that space.

## 6.1 Overview of Contributions

Our hypothesis at the outset was: *Hypothesis: An exploration of three distinct sub-genres within the coding games design space will 1) help produce more coding games, 2) yield novel ways of incorporating into coding games things we know gamers already enjoy, and 3) yield principles that will guide designers in creating future coding games.*

We have investigated three distinct sub-genres, each with a different level of integration between coding and gameplay. In this process, we have advanced the state of knowledge in each area individually, as well as in the domain of coding games as a whole.

### 6.1.1 Coding in a Game

*Contribution within this sub-genre*. Our main contributions are 1) to explore a richer set of metaphors for programmable objects, and 2) to explore the potential of a competitive multiplayer mode for educational purposes. Most previous games in this genre have been single player games, and they have employed a simple metaphor (i.e. program the bot). The research version of CodeSpells added a first-person viewpoint, placing the player into the world as an embodied (non-programmable) agent. This agent is a wizard that can enchant (program) objects. As such, the game employs a rich set of "magic" metaphors that are deeply rooted in our culture – largely as a result of seminal works of fantasy fiction, such as *The Lord of the Rings* and *Harry Potter*.

Consumers of such works of fiction have come to associate various ideas with magic, and these ideas have pedagogical utility within the domain of coding. Harry Potter, for example, must go to school to become a wizard – suggesting that magic (like coding) is an academic subject. Magic is an arcane art whose rules are not always clear at the outset. Magicians often fail (sometimes with humorous results) when they are first learning. All of these cultural expectations come to the forefront of the player's gaming/educational experience because of the prominence of the wizard/magic metaphor in CodeSpells. Because these expectations can have a profound effect on the player's educational experience, we hope that educational game designers will take as much care when selecting their game's governing metaphors as when they are designing the game itself.

On a different note, we have also explored the emergent properties of competitive multiplayer within the game of CodeSpells. Much like the competitive multiplayer communities of chess and StarCraft II, we have found that competition catalyses a kind of "adjacent" educational space – a space outside of the game itself, but which is nonetheless full of educational resources and rich with educational opportunity. Until there are significant advances in the field of artificial intelligence, the challenge of besting a fellow human being in competition will continue to be one of the most cognitively demanding tasks available to us. Human beings make for clever, creative, challenging opponents. They are capable of working long hours to derive new and better ways of winning. This results in a kind of intellectual "arms race", where all members of the competitive community are simultaneously working together to figure out how to beat each other (a kind of paradox of cooperation and competition). With the multiplayer version of CodeSpells, we have shown that such a community can have educational byproducts in the field of coding, software engineering, team management, etc.

*What this sub-genre tells us about the field as a whole.* The C[in]G sub-genre is more seamlessly integrated than the C[for]G sub-genre, but less so than the C[as]G sub-genre.

Compared with C[for]G systems, a C[in]G system affords the designer more control over the metaphors and how they shape the educational experience. A more integrated system also has its potential benefits in terms of "flow" and "immersion". That is, a player will probably be more immersed in a C[in]G system because they are never required to mode shift to a separate piece of software (as with C[for]G systems). In other words, tighter integration makes it easier to control the player's interaction and perceptions between the C part and the G part of the system.

However, compared with C[for]G systems, a C[in]G system involves a larger design burden. Integration is a design constraint that C[in]G systems maintain, while C[for]G systems relax. The cost of maintaining this constraint should be weighed against the benefits of higher degrees of player immersion.

### 6.1.2 Coding as a Game

*Contribution within this sub-genre.* Our main contribution in this space was to offer the second example system (aside from Toon Talk) within it. In doing so, we have further proven that the space is viable. We have also shown that complete novices can use such a system to perform algorithm-writing tasks involving map and fold constructs. And we have shown that the integration between coding and game can be so tight that the player does not realize they are writing code, even though they are.

*What this sub-genre tells us about the field as a whole.* Reflecting on this sub-genre in relation to the others leaves us feeling that considerably more work would need to be done before a system in this space could constitute a blockbuster coding game. The main blocker is that the game goes to such lengths to disguise the coding aspect that 1) the game's mechanics (recursion and conditionals) are strange and unfamiliar to gamers, and 2) the coding doesn't resemble coding and thus raises the question of whether skill at the game would transfer to coding skill in the real world. Neither of these characteristics are unique to our system, but rather are intrinsic to the sub-genre itself.

Even so, this gives us evidence to argue against the paradigm that currently dominates the educational games research community: seamless integration. In designing our C[as]G system, we took seamless integration to its logical extreme, and didn't like where we ended up. This allows us to recommend to future designers that they consider less integrated kinds of coding games. Seamlessness lies on a spectrum, after all. Many C[in]G systems exhibit qualities of seamlessness – e.g. CodeSpells contains an in-game IDE that the player must mode switch between. However, that IDE is presented as a spellbook, so although there is a mode switch, the player never has to stop being a wizard. Indeed, perhaps future work can give us a more nuanced examination of what exactly a "seam" is: perhaps in CodeSpells the seam exists at the level of "flow" [Che07] (because it divides the user's gameplay into two distinct types), but perhaps there is no seem at the level of "fantasy" [Mal81b].

This is not to say that the C[as]G sub-genre is a dead end – just that there are significant problems to be solved within it[1]. The potential payoff in the other sub-genres may simply be greater at this time.

### 6.1.3 Coding for a Game

*Contribution within this sub-genre.* This space has historically been explored from the perspective of building novice-friendly IDEs for making a variety of games (Alice, Scratch, Greenfoot, etc.) Our contribution is to create an IDE that is specifically for modding *one* game – Minecraft. Because Minecraft is wildly popular, we hoped that this popularity would "rub off on" our system. This hypothesis turned out to be accurate and suggests that the loss of generality is warranted, and that the use of a popular game drives recruitment.

Of all the systems we have built, LearnToMod has been the most straightforward to prototype and to commercialize. Only time will tell whether it will ever attain the label "blockbuster", but we are encouraged by the early success.

*What this sub-genre tells us about the field as a whole.* Compared with other systems, LearnToMod suggests that the use of an existing game relieves the designer of the considerable burden of designing a game from the ground up (though they are still free to do so if they wish). And by dis-integrating the educational component from the gameplay, a kind of modularity is achieved (at the expense of "seamless integration"): the designer is better able to focus on the pedagogical aspects of the education part, without

[1]Since C[as]G systems are also visual programming languages, perhaps some of the challenges are the same – e.g. designing good visual representations for coding constructs and runtime operations [Bur98].

having to simultaneously consider gameplay-related constraints. Whereas such a dis-integrated approach flies in the face of the prevailing paradigm in seamlessly integrated educational games research, we hope that our successes will help broaden the horizons of our fellow researchers and designers. After all, our most successful and most educational system was the one that was the least integrated.

## 6.2 Design space

Our experiences building and studying the three systems above help us identify distinct axises that describe points in the coding games design space. There are axises that are particularly important, two of which pertain to the skills required of an individual or team when designing an educational coding game, and two of which pertain to how the end-user experiences the educational coding game. All four are critical for designers to take into consideration.

As a design process, we have two distinct axises: 1) Integrated development environment and programming language (IDE/PL) skills required, and 2) game design skills required. As an end-user experience, we have two other axises: 1) degree of integration of game and coding, and 2) mainstream buy-in.

These design process dimensions and the end-user dimensions are two sides of the same coin. A designer (or team) wishing to achieve a particular degree of integratedness or mainstream appeal will require a particular level of skill in game design and/or IDE/PL. And conversely, teams with different strengths in IDE/PL versus game design will be differently equipped to reach points in the design space that have different levels of integration and/or mainstream appeal. We discuss each dimension individually in the following sections. But it is worth keeping in mind that they are interconnected.

### 6.2.1 IDE/PL Skills Required

Designing an IDE and/or a programming language is no easy task (even when *not* embedded in a game). Both a writetime and runtime environment must be crafted – a modality in which the programmer writes and edits code, as well as a modality in which the programmer runs the code and obtains output. Syntax and runtime errors must be effectively communicated to the programmer. In many cases, this requires performing static analysis on the code (e.g. to detect syntax errors or type errors), and runtime errors must be traced back to the relevant locations in the code.

All of these concerns have been solved for expert-level IDEs, but not necessarily for novice IDEs. Even Scratch – a state of the art novice IDE does not have support for realtime debugging or the display of runtime error messages. No routine static analysis (e.g. type checking) is performed either. Indeed, how to implement these expert-level bells and whistles in a novice-friendly way is an open question in the field of novice-friendly IDEs.

All of these difficulties exist independently of the task of integrating an IDE with a game. Whether a project requires building an IDE from scratch or using an off-the-self solution, a designer needs to have a solid understanding of how user-interfaces for programmers work, as well as how these user-interfaces must be changed to support novices.

A team or individual that is weak in its IDE and programming language skills would be advised to gravitate toward a C[for]G solution – where using off-the-shelf solutions becomes much easier. Our research prototype of LearnToMod, for example, was built with minimal IDE design burden. The only requirement was an understanding of how JavaScript executes within Java's virtual machine – so that we could pipe JavaScript written in the Ace editor into a series of pre-processors and, ultimately, the JavaScript runtime. Although this was non-trivial, it was by far the easiest IDE-related task we performed across our three projects.

CodeSpells was more difficult, and The Orb Game was the most difficult of all. In the research version of CodeSpells, we had to build a writetime and runtime bridge between the game and Java's compiler and runtime. We had to figure out how to display syntax and error messages in the in-game text editor, and we had to figure out how to display runtime errors when a spell failed due to runtime problems. In the commercial version of CodeSpells, we used Blockly, which required us to understand how Blockly compiled into JavaScript, and how JavaScript could be pre-processed and passed along and evaluated by the Unity runtime.

For The Orb Game, we designed an entirely unique programming language, defined by in-game avatar movements. We had to be able to compile those in-game operations into a format that we could execute. We had to provide a modality for players to be able to write code by moving their avatar, and also a modality for them to be able to watch their code execute (i.e. a debugger) for understanding their code's runtime behaviour. And on top of all of this, we made the system a programming-by-demonstration system, which meant that we had to perform type abstractions on in inputs to a user's program, thus deriving a more abstract version of the program than that which the user was actually writing. This was non-trivial indeed.

In other words: *A C[for]G system requires the least technical skill with regard to IDE-design and programming language design because an off-the-shelf IDE can be leveraged without modification. A C[in]G system requires more technical know-how because the IDE must be rendered in-game, often meaning that many IDE features must be reimplemented within the context of the game engine (not to mention its set of UI components and metaphors). A C[as]G system is the most non-trivial of all because it requires complete reimplementation of an IDE and programming language as game constructs.*

### 6.2.2 Game Design Skills Required

The other half of the game+coding puzzle is the game design aspect. The challenges of game design "in the abstract" are myriad but boil down to the fact that game design is essentially an art, not a science. As with any art, there are a few guiding principles, but no hard-and-fast rules. Games with "good graphics" (e.g. Skyrim) seem to sell better and be more enjoyable – but what makes graphics "good" is a subjective question, and there are many exceptions to this rule (e.g. Minecraft today, and the old text-based RPGs of a few decades ago). Games with a "good story" (e.g. Portal) tend to be more meaningful – but again, the notion of "good" is hard to pin down, and there are many games with no story that are still enjoyable.

The fact of the matter is that games are difficult to design, and no one knows for certain beforehand whether or not the game will be an enjoyable experience. In other fields, the answer to this problem is iterative design – but this is less feasible in games than in other fields. Games are complex works of art, with embedded stories, graphics, user interfaces, music, mechanics, etc. Whereas various aspects of the game can be marginally improved based on user feedback after the game has been completed, the totality of the experience is difficult to change. Because of this, game designers (whether in industry or in academia) play a kind of guessing game. They do their best to try to hit a high-point in the space of possible games. Some of them succeed. Many of them fail. The process of trying to craft a fun, popular game is like searching for a needle in a hay stack, where the search is guided only by the fuzziest of heuristics. As [IFH10] observes, such results are often published as blogs on sites like Gamasutra.

In our experience, LearnToMod was the easiest system to design, due to the fact that we didn't have to design a game at all; instead we hitched an IDE to a game that had already proven itself to be a high-point in the space of all possible games. CodeSpells was considerably harder because we had to craft our own artistic assets, our own game logic, etc. And The Orb Game was by far the hardest because we had to design our own game, while also making sure that it remained a Turing complete programming environment.

In other words: *A C[for]G system requires little to no game design skill. A C[in]G system requires building a new game from scratch, meaning that it requires at least as much skill as designing a non-educational game. And a C[as]G system requires even more skill than would be required for a "normal" game because of the added design constraints – that the game seamlessly integrate coding.*

### 6.2.3 Degree of integration

A design dimension that profoundly affects the end-user experience is the degree to which coding and gameplay are integrated. The tendency in academia is to prejudice tighter integration over looser integration. This, we feel, is a mistake.

It's unclear exactly how the idea of "seamless integration"/"chocolate-covered broccoli" came to take such a strong hold of educational games research community. The earliest record we could find was Amy Bruckman's denouncement of chocolate-covered broccoli in this manifesto [Bru99], where she cites Math Blaster as an exemplar of a poorly integrated game. Advancement of the idea of "seamless integration" (and the synonymous avoidance of "chocolate-covered broccoli") has been cited many times since [FEG13, MWM12a, AVW07, HA11, Pre04, HA11, IFH10], and always (in our opinion) without convincing scientific evidence to back it up. It is as if the aesthetic principle has become a dominant guiding force in scientific research based on the attractiveness of the aesthetics alone.

We cannot go so far as to say that seamless integration is bad (and that the chocolate-covered broccoli approach is good). But the success of LearnToMod – a highly *dis*integrated game – where the educational content is literally in a different piece of software from the game – suggests that lack of integration has its merits, too. In our case, the merit is that it allowed us to leverage an already-existing popular game, without changing it, and to simply "bolt on" an educational component with scarcely any effort spent on integrating it.

Although our experience is that this strategy has been wildly successful, the very idea is almost heretical in the current anti-chocolate-broccoli research climate. This is sad because it means that the design space of disintegrated educational games is not being sufficiently explored. Again, it reminds one of the denouncement of early heliocentric theories of the solar system, which (although correct) violated the religious and aesthetic principles implied by the predominant geocentric model. We will admit that the idea of seamless integration has its charm; and chocolate-covered broccoli does *sound* disgusting. Aesthetics aside, though, our own experiences undermine the scientific utility holding such beliefs so strongly.

Our recommendation, then, to educational game designers is to consider the full range of possible integrations before immediately showing a preference for the "fully integrated" side of the spectrum. Dis-integration allows the educational content to be made explicit, rather than hidden. Here are several (untested) reasons why this *might* be a good thing:

- *Transfer*. When the educational content is presented in a purely "academic", undisguised manner, it is plausible that students will be more successful when transferring that knowledge to other academic settings. It is plausible to assume that when content is explicitly disguised (so that it integrates with a game), the player's ability to transfer the knowledge to the real-world is inhibited.

- *Metacognition*. Setting the educational content apart from the game may encourage the player to reflect on the content in ways that they might not reflect on games.

Indeed, our work with multiplayer CodeSpells shows that players are comfortable leaving the game in order to learn more about playing the game. They reflect on this material in forums, with videos, with out-of-game drills, etc. The very act of leaving the game to do this reflection may put people in a state of mind that is more conducive to reflection.

- *More degrees of freedom.* As a designer, the viable design space is larger when it isn't constrained by the idea of integration-at-all-costs. When we held this constraint at all costs, we ended up with a strange game that wasn't necessarily educational in the ways we wanted it to be (The Orb Game). By relaxing this constraint and bolting an automated tutoring system onto an existing game, we produced a system that was more fun and more educational.

### 6.2.4 Mainstream buy-in

Mainstream buy-in is the reason a designer would choose to build an educational coding game in the first place. Games are popular. The problem is that it's hard to build a popular game.

Building a game from scratch with strange mechanics (e.g. The Orb Game) is a very risky approach. Building a game from scratch that looks and feels like other popular games (e.g. as CodeSpells looks and feels like Skyrim) is still risky, but perhaps less so. Building onto an existing game that is already extremely popular (e.g. as LearnToMod builds onto Minecraft) is the lowest risk approach of all. And this is not merely a theory: our results with LearnToMod show that recruitment numbers an intrinsic motivation are high – in spite of the fact that we have added educational content to the video game.

Mainstream buy-in is not something that system designers have direct control over. Rather, it is an emergent property that one hopes for. However, the choice of sub-genre does have a direct bearing on one's chances of reaching a point in the design space with high levels of mainstream buy-in.

On this note, we have a word of caution for other designers: Not to stop at the surface-level idea that "games are popular". All games are different. Different game mechanics are appealing in different ways. Some games are popular because they allow the user to "zone out" and stop thinking (e.g., Candy Crush). Some games are popular because they require intense concentration and years of practice (e.g., StarCraft II). In other words, "mainstream buy-in" is a complex, multi-faceted thing. Simply making an educational system that involves a game and hoping for mainstream buy-in may work, but isn't what we would recommend. Instead, we would encourage a nuanced examination of what aspects of the game are intended to appeal, and how those aspects related to the educational component of the system.

### 6.2.5 Development Case Studies

We now give short case studies to further articulate how the concepts presented in this chapter informed our process.

*CodeSpells - Research version.* CodeSpells was difficult to build because it required building two things: a game and an embedded IDE. We couldn't leverage an existing IDE (like Eclipse) because we were designing the game in Unity, and we needed the IDE to be rendered in-game. So we were left with two difficult design tasks. Because the research version of CodeSpells used Java as the language of instruction, we had to build a Java IDE – which has its own inherent difficulties, like support for separate class files, display of compiler errors, etc. Still, at the end of the day, many good Java IDEs exist, so we at least had many reference systems to aid our design.

*CodeSpells - Commercial version.* Although the authors did not build the commercial version of CodeSpells, the author did manage the ThoughtSTEM employees who built it. After observing the ongoing design process for the commercial version, we can comment on the unique challenges.

To increase mainstream buy-in, the decision was made not to use Java[2]. Instead, the designers opted for Blockly, a user-friendly visual programming language much like Scratch. Blockly is implemented in web technologies – HTML, CSS, and JavaScript – which are not native to Unity. However, embedding them in Unity was facilitated by a recently developed technology called CoherentUI – which allows a browser to be rendered in-game. Thus, the development of the IDE was surprisingly easy.

Developing the rest of the game, however, was much harder. The overarching opinion of the designers was that the game should be fun – not overtly educational. Again, this was to increase mainstream buy-in. The idea was, first and foremost, to make an experience that was fun to play – and only an educational opportunity if the player explicitly wanted it to be. In this regard, the designers were heavily inspired by Minecraft – which is currently being used in thousands of classrooms across the world, and is being used to teach literally every subject – yet the game was never designed to be "educational". It gets its educational value from the fact that it is a highly flexible platform for creativity, and teachers have found ways of incorporating it into their curriculum in numerous ways. So too was CodeSpells intended to be a game first, and educational second.

For example, the player is not required to write any code if they don't want to. They are given a list of pre-written spells. They can explore the environment and manipulate the world with only those spells, if that's what they wish to do. The game has not been released for long enough to draw many conclusions, but the current hypothesis (and goal) is that 90% of users will be content simply to play the game, whereas 10% will be interested in modifying the spells and creating new ones that they will then share

[2]As the reader may recall, Java was chosen in the research version to strengthen the theoretical result: i.e., "if we can make Java appealing, we can make anything appealing".

with the rest of the community. A similar ratio of consumers to producers can be seen in the Minecraft modding community and even in spaces like YouTube. There are always many more consumers and producers. This is a feature of the “mainstream”: for better or worse, most people tend to consume more than they produce.

Because of these design goals (and the existence of Blockly and CoherentUI), the entire design burden was reduced to the design of the game: Creating artistic assets, coding the game logic, designing the spellcrafting API, etc. A single developer and artist were able to assemble a Beta version about 9 months after the Kickstarter campaign was successfully funded. It certainly wasn’t trivial by any means, but being able to leverage open source technology for the IDE and visual programming language made the task much easier.

*The Orb Game*. In The Orb Game, the coding part and the game part are one in the same. So one might guess that this made things easier than if they had to be designed separately. However, it was precisely the activity of combining the two into one that made the design work so difficult.

For every coding construct necessary for list processing and Turing completeness, we had to find a way to represent that construct as a game mechanic. Sometimes that was straightforward – e.g. with basic operations on lists (pop, concat, etc.). But with conditionals and recursion, the mechanics were not straightforward. We did our best to keep the interface looking like a Mario-style platformer, but sometimes the behaviour was necessarily complex and un-gamelike, not to mention difficult to design. Overall, the project was difficult to complete and resulted in something that wasn’t very fun, nor very useable. Players could perform reasonably well on the benchmarks, but it would take a lot more work to make the game commercially viable. And even then, we are pessimistic that the game would attract a userbase.

*LearnToMod - Research version*. This prototype was perhaps the easiest of all to build. The game of Minecraft already existed, so we didn't have to build a game from scratch. And even the tools for modding it had already been invented by the open source community. The Scriptcraft mod had been created the year before, allowing people to mod Minecraft with JavaScript. So all we had to do was create a JavaScript IDE. This too was easy, since we were able to leverage the open-source Ace editor – a web-based JavaScript. And because this was a C[for]G system, we didn't even have to embed the IDE into Minecraft. All we had to do was pipe the JavaScript from the IDE into Minecraft – a trivial system to build.

*LearnToMod - Commercial version.* Although the authors did not build the commercial version of LearnToMod, one of the authors did manage the ThoughtSTEM employees who built it. After observing the ongoing design process for the commercial version, we can comment on the unique challenges involved.

The main challenge here was to build a more robust IDE – one that supported code sharing, automated tutoring, and an online community. The IDE was implemented as a Ruby on Rails app with a Blockly-based front-end and took the developers several months to complete. As outside observers, we have noticed an interesting symmetry between the design of the commercial versions of CodeSpells and LearnToMod: With CodeSpells, the majority of the design work was in building the game; with LearnToMod, the majority of the design work was in building the IDE.

## So which was the easiest?

While there were challenges in each system, the one whose prototype was easiest to assemble was LearnToMod (the research version), since so much of the work was already done. We would venture to say that this is a major benefit of choosing the C[as]G strategy: The game is already built.

## 6.3 Discussion of Other Contributions

Although this thesis gives much weight to design insights, we have made other contributions that are noteworthy. This section lists some of the non-design-related contributions.

### 6.3.1 Gender

In Chapter 1, we discussed the gender disparity issue in computer science. About 44% of gamers are female [esa15], which is twice the representation than the 21% of women who graduate with a CS degree. It was difficult to acquire finegrained demographic data about players of particular games – e.g. Minecraft and StarCraft. Our StarCraft II respondents were almost overwhelmingly male (over 90%). However, this was also the case for this online survey of Minecraft players [Can12] – where the population was also over 90% male. This is odd because our own recruitement numbers for Minecraft show that we are recruiting 30% women (even higher than our 10% Scratch recruitment numbers). Does this mean that there are actually many more female Minecraft players than the aforementioned survey would seem to indicate? Might this also be true for StarCraft II then? Perhaps male gamers are more likely to fill out online surveys about games. Or perhaps the Minecraft community has changed since the survey was conducted in 2012. The one thing we can be sure of is that our findings would suggest much higher diversity in the Minecraft community than [Can12] finds.

In terms of future work: It has been shown that games may be more beneficial than lectures for female students [SBGH04][3]. It would be beneficial to investigate if this holds true for coding education – especially now that we are recruiting more female students. Furthermore, although our 30% recruitment numbers represents an

[3]This was also true for male students but to a lesser degree

improvement in diversity over the 21% college graduation rates, we would still like to 1) improve this number further, and 2) investigate long-term rentention of women in the Minecraft condition.

### 6.3.2 Player communities

In Chapter 2, we presented a survey of StarCraft II and chess communities. We used that data to motivate crafting a community around CodeSpells. However, it is worth noting that our discoveries from that survey also contribute to the body of knowledge about player habits and attitudes. As mentioned in the previous section about gender, it is difficult to find empirical studies that reveal even basic information about sub-communities of gamers (like how many girls play Minecraft). We quantified the amount of supplemental training that varoius player populations engage in (71% of StarCraft II players do some form of training more than one hour per week, sometimes much more). We characterized these communities as probably having a growth mindset. We know of no prior study that has sought to determine the general (Dweckian) mindset for a group of gamers. Papert noted in the 1990s that gamers seemed to have surprisingly accurate learning-theoretical understanding of how to improve at video games (better perhaps than their understanding of how to improve in other subjects) [pap15]. So our result gives empirical weight to Papert's observation – at least in the matter of growth mindset. The community believes that practice is important; those practices are metacognitive in nature; and they spend a considerable amount of time doing that kind of practice.

### 6.3.3 Games and pair-playing

When sharing a single computer and asked to work together, the Minecraft students were much more likely than the Scratch students to become distracted or to put their heads down [4]. This was somewhat of a surprising result, given that a quick Youtube search will turn up thousands of Minecraft play sessions with millions of views, demonstrating that watching other people play Minecraft is a popular activity. Why should this be different in a classroom setting? Anecdotally: One young man would put his head down on the desk and pretend to sleep every time it was his partner's turn at the computer. Something about the social and/or physical setting seems to alter the way players are willing to observe another individual's play sessions. We know of no prior studies that investigate attitudes and behaviours around over-the-shoulder participation in video games. Our result shows a difference in how such participation occurs around a popular video game versus another type of software (an IDE).

## 6.4 Concluding Remarks

This thesis and the work it reports on have been impactful in numerous ways:

- We have investigated a prevailing design principle – seamless integration – and discovered a non-seamless (seamful) systems can also be intrinsically motivating to students.
- We have presented a seamful IDE+Game pattern that can be repeated with other games or other IDEs with relatively low design burden.
- We have shown that competitive multiplayer can be used in coding games as way of giving players pedagogically productive in-game activities – which may require less design work than creating a repository of singleplayer activities.
- We have created three novel coding games, thus expanding the ecosystem of coding games that work on modern hardware.

[4] Anecdotally, the girl partners actually collaborated quite well on a single computer. It was the male partners who had much more trouble.

- We have partitioned the coding games design space into three sub-genres C[as]G, C[in]G, and C[for]G.
- We have built three systems, one in each sub-genre, advancing the state-of-the-art in each.
- We have set the stage for two commerical products – CodeSpells and LearnToMod. Both have been externally funded by KickStarter and an NSF SBIR Grant.
- Ongoing academic research on LearnToMod is being performed by the University of Maine (in a recent STEM+C grant).

Our hypothesis was that an examination of three sub-genres within the coding games design space would help create new coding games, new ways of making those games better resemble mainstream games, and new design guidelines for creating coding games. In pursuit of this goal, we have advanced the state of the art in coding games on several fronts and contributed to launching two commercial products. With the ever-growing importance of coding skills in today's economy, the quest for the blockbuster coding game becomes increasingly important too. This thesis work is a significant step toward this goal.

# Bibliography

[AKR$^{+}$02] Gregory Aist, Barry Kort, Rob Reilly, Jack Mostow, and Rosalind Picard. Experimentally augmenting an intelligent tutoring system with human-supplied capabilities: Adding human-provided emotional scaffolding to an automated reading tutor that listens. In *Proceedings of the 4th IEEE International Conference on Multimodal Interfaces*, ICMI '02, pages 483–, Washington, DC, USA, 2002. IEEE Computer Society.

[AR14] Deborah J. Armstrong and Cindy K. Riemenschneider. The barriers facing women in the information technology profession: An exploratory investigation of ahuja's model. In *Proceedings of the 52Nd ACM Conference on Computers and People Research*, SIGSIM-CPR '14, pages 85–96, New York, NY, USA, 2014. ACM.

[AVW07] Nicoletta Adamo-Villani and Kelly Wright. Smile: An immersive learning game for deaf and hearing children. In *ACM SIGGRAPH 2007 Educators Program*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[AWP99] Wilfried Admiraal, Theo Wubbels, and Albert Pilot. College teaching in legal education: Teaching method, students' time-on-task, and achievement. *Research in Higher Education*, 40(6):687–704, 1999.

[Bad90] Joanne M. Badagliacco. Gender and race differences in computing attitudes and experience. *Social Science Computer Review*, 8(1):42–63, 1990.

[bat12] http://us.battle.net/sc2/en/blog/2053471, August 2012.

[Bau12] Eric B. Bauman. *Game-Based Teaching and Simulation in Nursing and Health Care*. Springer, 2012.

[BBGP09] F. Bellotti, R. Berta, A. De Gloria, and L. Primavera. Enhancing the educational value of video games. *Comput. Entertain.*, 7(2):23:1–23:18, June 2009.

[BBRB12] Victoria Simpson Beck, Stephanie Boys, Christopher Rose, and Eric Beck. Violence against women in video games: A prequel or sequel to rape myth acceptance? *Journal of Interpersonal Violence*, 27(15):3016–3031, 2012.

[BCD89] J.S. Brown, A. Collins, and P. Duguid. Situated cognition and the culture of learning. *Educational Researcher*, 18(1):32, 1989.

[BCKW04] Ryan Shaun Baker, Albert T. Corbett, Kenneth R. Koedinger, and Angela Z. Wagner. Off-task behavior in the cognitive tutor classroom: When students ”game the system”. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI ’04, pages 383–390, New York, NY, USA, 2004. ACM.

[BDWK10] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott 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*, CHI ’10, pages 513–522, New York, NY, USA, 2010. ACM.

[Bla11] Jonathan D. Blake. Language considerations in the first year cs curriculum. *J. Comput. Sci. Coll.*, 26(6):124–129, June 2011.

[bls15] http://www.bls.gov/ooh/computer-and-binformation-technology/software-developers.htm, September 2015.

[BM05] Theresa Beaubouef and John Mason. Why the high attrition rate for computer science students: Some thoughts and observations. *SIGCSE Bull.*, 37(2):103–106, June 2005.

[BoDitSoLoLRP00] John D. Bransford, National Research Council Etats-Unis. Committee on Developments in the Science of Learning., National Research Council Etats-Unis. Committee on Learning Research, and Educational Practice. *How people learn : brain, mind, experience, and school.* National Academy Press, 2 edition, September 2000.

[BP02] Sasha A. Barab and Jonathan A. Plucker. Smart people or smart contexts? cognition, ability, and talent development in an age of situated approaches to knowing and learning. *Educational Psychologist*, 37:165–182, 2002.

[Bru99] Amy Bruckman. Can educational be fun? In *presented at the Game Developers Conference '99, San Jose, CA.*, 1999.

[BS08] Andrew Begel and Beth Simon. Struggles of new college graduates in their first software development job. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 226–230, New York, NY, USA, 2008. ACM.

[Bur98] Margaret M. Burnett. Challenges and oppurtunities visual programming languages bring to programming language research. In *Proceedings of the 7th International Conference on Compiler Construction*, CC '98, pages 188–, London, UK, UK, 1998. Springer-Verlag.

[Can12] Alessandro Canossa. Give me a reason to dig: Qualitative associations between player behavior in minecraft and life motives. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, pages 282–283, New York, NY, USA, 2012. ACM.

[Car89] John B. Carroll. The carroll model: A 25-year retrospective and prospective view. *Educational Researcher*, 18(1):26–31, 1989.

[CBP13] Yaniv Corem, Naor Brown, and Jason Petralia. Got skillz?: Player matching, mastery, and engagement in skill-based games. In *Proceedings of the First International Conference on Gameful Design, Research, and Applications*, Gamification '13, pages 115–118, New York, NY, USA, 2013. ACM.

[CG04] Matthew Chalmers and Areti Galani. Seamful interweaving: Heterogeneity in the theory and design of interactive systems. In *Proceedings of the 5th Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques*, DIS '04, pages 243–252, New York, NY, USA, 2004. ACM.

[CH11] Gifford Cheung and Jeff Huang. Starcraft from the stands: understanding the game spectator. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, pages 763–772, New York, NY, USA, 2011. ACM.

[Che07] Jenova Chen. Flow in games (and everything else). *Commun. ACM*, 50(4):31–34, April 2007.

[CHK$^+$93] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[CL06] Christine Cross and Margaret Linehan. Barriers to advancing female careers in the high tech sector: empirical evidence from ireland. *Women in Management Review*, 21(1):28–39, 2006.

[cod12] codespells.blogspot.com, August 2012.

[cod14] http://code.org/stats, September 2014.

[cs 15] http://www.nsf.gov/statistics/seind12/append/c2/at02-18.pdf, September 2015.

[CZP14] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. Identifying challenging cs1 concepts in a large problem dataset. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 695–700, New York, NY, USA, 2014. ACM.

[Dal06] Nell B. Dale. Most difficult topics in cs1: Results of an online survey of educators. *SIGCSE Bull.*, 38(2):49–53, June 2006.

[DCP06] Wanda P Dann, Stephen Cooper, and Randy Pausch. *Learning to Program with Alice, Brief Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[DDE$^+$12] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Nathaniel Mote, Brian Walker, Seth Cooper, Timothy Pavlik, and Zoran Popović. Verification games: making verification fun. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 42–49, New York, NY, USA, 2012. ACM.

[deg65] A. D. degroot. *Thought and Choice in Chess*. Mouton, The Hague, The Netherlands, 1965.

[Dic11] Matthew Dickerson. Multi-agent simulation and netlogo in the introductory computer science curriculum. *J. Comput. Sci. Coll.*, 27(1):102–104, October 2011.

[DKP12] Laura Dabbish, Robert Kraut, and Jordan Patton. Communication and commitment in an online game team. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, CHI '12, pages 879–888, New York, NY, USA, 2012. ACM.

[DKR99] Edward L. Deci, Richard Koestner, and Richard M. Ryan. A meta-analytic review of experiments examining the effects of extrinsic rewards on intrinsic motivation. *Psychological Bulletin*, 125:627–668, 1999.

[Don07] Mary Jo Dondlinger. Educational Video Game Design: A Review of the Literature. 2007.

[dRWT03] Michael de Raadt, Richard Watson, and Mark Toleman. Language tug-of-war: industry demand and academic choice. In *Proceedings of the fifth Australasian conference on Computing education - Volume 20*, ACE '03, pages 137–142, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[Dun11] Sean C. Duncan. Minecraft, beyond construction and survival. *Well Played*, 1(1):1–22, January 2011.

[Dwe07] Carol Dweck. *Mindset: The New Psychology of Success*. Ballantine Books, 2007.

[Eag09] Michael Eagle. Level up: A frame work for the design and evaluation of educational games. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09, pages 339–341, New York, NY, USA, 2009. ACM.

[EB12] Michael John Eagle and Tiffany Barnes. A learning objective focused methodology for the design and evaluation of game-based tutors. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 99–104, New York, NY, USA, 2012. ACM.

[EFG13a] Sarah Esper, Stephen R. Foster, and William G. Griswold. 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, pages 305–310, New York, NY, USA, 2013. ACM.

[EFG13b] Sarah Esper, Stephen R. Foster, and William G. Griswold. 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, pages 305–310, New York, NY, USA, 2013. ACM.

[EFG13c] Sarah Esper, Stephen R. Foster, and William G. Griswold. 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, pages 305–310, New York, NY, USA, 2013. ACM.

[EFG$^+$14] Sarah Esper, Stephen R. Foster, William G. Griswold, Carlos Herrera, and Wyatt Snyder. Codespells: Bridging educational language features with industry-standard languages. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Calling '14, pages 05–14, New York, NY, USA, 2014. ACM.

[EG14] Barbara Ericson and Mark Guzdial. Measuring demographics and performance in computer science education at a nationwide scale using ap cs data. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 217–222, New York, NY, USA, 2014. ACM.

[EKTr93] K. Anders Ericsson, Ralf Th. Krampe, and Clemens Tesch-romer. The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, pages 363–406, 1993.

[ENS06] Magy Seif El-Nasr and Brian K. Smith. Learning through game modding. *Comput. Entertain.*, 4(1), January 2006.

[Eri08] K. Anders Ericsson. *Attaining Excellence Through Deliberate Practice: Insights from the Study of Expert Performance*, pages 4–37. Blackwell Publishers Ltd, 2008.

[esa15] http://www.theesa.com/wp-content/uploads/2015/04/esa-essential-facts-2015.pdf, September 2015.

[EWF$^+$14] Sarah Esper, Samantha R. Wood, Stephen R. Foster, Sorin Lerner, and William G. Griswold. Codespells: How to design quests to teach java concepts. *J. Comput. Sci. Coll.*, 29(4):114–122, April 2014.

[FCJ04] T. Flowers, C.A. Carver, and J. Jackson. Empowering students and building confidence in novice programmers through gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, pages T3H/10 – T3H/13 Vol. 1, oct. 2004.

[FEG13] Stephen R. Foster, Sarah Esper, and William G. Griswold. From competition to metacognition: Designing diverse, sustainable educational games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 99–108, New York, NY, USA, 2013. ACM.

[FG04] Andrea Forte and Mark Guzdial. Computers for communication, not calculation: Media as a motivation and context for learning. In *Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 4 - Volume 4*, HICSS '04, pages 40096.1–, Washington, DC, USA, 2004. IEEE Computer Society.

[Fis98] Robert Fisher. Thinking about thinking: Developing metacognition in children. *Early Child Development and Care*, 141:1–15, Feb 1998.

[FP09] Katrina Falkner and Edward Palmer. Developing authentic problem solving skills in introductory computing classes. *SIGCSE Bull.*, 41(1):4–8, March 2009.

[FW80] Wayne C. Fredrick and Herbert J. Walberg. Learning as a function of time. *The Journal of Educational Research*, 73(4):pp. 183–194, 1980.

[gam12] http://gamereplays.org, August 2012.

[GB13] Ben Gibson and Tim Bell. Evaluation of games for teaching computer science. In *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*, WiPSE '13, pages 51–60, New York, NY, USA, 2013. ACM.

[GC06] Fernand Gobet and Neil Charness. Expertise in chess. In Fernand Gobet and Neil Charness, editors, *Chess and games. Cambridge handbook on expertise and expert performance*, pages 523–538. Cambridge University Press, Cambridge, Massachusetts, USA, 2006.

[Gee03] James Paul Gee. What video games have to teach us about learning and literacy. *Comput. Entertain.*, 1(1):20–20, October 2003.

[Gee05a] J. P. Gee. What would a state of the art instructional video game look like. *Innovate Journal of online education*, 1(6), 2005.

[Gee05b] James Paul Gee. Learning by design: Good video games as learning machines. *E-Learning and Digital Media*, 2(1):5–16, 2005.

[Gee07] James Gee. Learning and games. In Katie Salen, editor, *The Ecology of Games: Connecting Youth, Games, and Learning*, pages 21–40. MIT Press, 2007.

[GM08] Randy J. Gallant and Qusay H. Mahmoud. Using greenfoot and a moon scenario to teach java programming in cs1. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, pages 118–121, New York, NY, USA, 2008. ACM.

[GM11] John Gibson and David McKenzie. Eight questions about brain drain. *The Journal of Economic Perspectives*, 25(3):pp. 107–128, 2011.

[GmGmGc04a] Marco A. Gmez-martn, Pedro P. Gmez-martn, and Pedro A. Gonzlez-calero. Game-driven intelligent tutoring systems. In *In Entertainment Computing - ICEC 2004, Third International Conference, Lecture Notes in Computer Science*, pages 108–113. Springer, 2004.

[GMGMGC04b] MarcoA. Gmez-Martn, PedroP. Gmez-Martn, and PedroA. Gonzlez-Calero. Game-driven intelligent tutoring systems. In Matthias Rauterberg, editor, *Entertainment Computing ICEC 2004*, volume 3166 of *Lecture Notes in Computer Science*, pages 108–113. Springer Berlin Heidelberg, 2004.

[GMT14] Carina González, Alberto Mora, and Pedro Toledo. Gamification in intelligent tutoring systems. In *Proceedings of the Second International Conference on Technological Ecosystems for Enhancing Multiculturality*, TEEM '14, pages 221–225, New York, NY, USA, 2014. ACM.

[GS10] James Paul Gee and David Shaffer. Looking where the light is bad: Video games and the future of assessment. *EDge*, 2010.

[HA11] M P. Jacob Habgood and Shaaron E. Ainsworth. Motivating children to learn effectively: Exploring the value of intrinsic integration in educational games. *Journal of the Learning Sciences*, 20(2):169–206, 2011.

[HAB05] M. P. J. Habgood, S. E. Ainsworth, and S. Benford. Endogenous fantasy and learning in digital games. *Simulation and Gaming*, 36(4):483–498, 2005.

[HBG14] Michael James Heron, Pauline Belford, and Ayse Goker. Sexism in the circuitry: Female participation in male-dominated popular computer culture. *SIGCAS Comput. Soc.*, 44(4):18–29, December 2014.

[Hen07] Michi Henning. Api design matters. *Queue*, 5(4):24–36, May 2007.

[HK95] Nadeem U. Haque and Se-Jik Kim. "human capital flight": Impact of migration on income and growth. *Staff Papers (International Monetary Fund)*, 42(3):pp. 577–607, 1995.

[HMA13] Erik Harpstead, Brad A. Myers, and Vincent Aleven. In search of learning: Facilitating data analysis in educational games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 79–88, New York, NY, USA, 2013. ACM.

[HMAM14] Erik Harpstead, Christopher J. MacLellan, Vincent Aleven, and Brad A. Myers. Using extracted features to inform alignment-driven design ideas in an educational game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 3329–3338, New York, NY, USA, 2014. ACM.

[Hon98] Jason Hong. The use of java as an introductory programming language. *Crossroads*, 4(4):8–13, May 1998.

[HSCJ09] Michael S. Horn, Erin Treacy Solovey, R. Jordan Crouser, and Robert J.K. Jacob. Comparing the use of tangible and graphical programming languages for informal science education. In *Proceedings of the 27th international conference on Human factors in computing systems*, CHI '09, pages 975–984, New York, NY, USA, 2009. ACM.

[IFH10] Katherine Isbister, Mary Flanagan, and Chelsea Hash. Designing games for learning: insights from conversations with designers. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 2041–2044, New York, NY, USA, 2010. ACM.

[Jab07] Janusz Jablonowski. A case study in introductory programming. In *Proceedings of the 2007 international conference on Computer systems and technologies*, CompSysTech '07, pages 82:1–82:7, New York, NY, USA, 2007. ACM.

[JDM10] G.Tanner Jackson, KyleB. Dempsey, and DanielleS. McNamara. The evolution of an automated reading strategy tutor: From the classroom to a game-enhanced automated system. In Myint Swe Khine and Issa M. Saleh, editors, *New Science of Learning*, pages 283–306. Springer New York, 2010.

[JML11] David Kirsh Joshua M. Lewis, Patrick Trinh. A corpus analysis of strategy video game play in starcraft: Brood war. In T. Shipley L. Carlson, C. Hlscher, editor, *Proceedings of the 33rd Annual Conference of the Cognitive Science Society*, pages 687–692, 2011.

[JS05] Janis E. Jacobs and Sandra Denise. Simpkins. *Leaks in the pipeline to math, science, and technology careers / Janis E. Jacobs, Sandra D. Simpkins, editors*. Jossey-Bass San Francisco, 2005.

[KÏ0] Michael Kölling. The greenfoot programming environment. *Trans. Comput. Educ.*, 10(4):14:1–14:21, November 2010.

[Kar84] Nancy Karweit. Time-on-task reconsidered: Synthesis of research on time and learning. *Educational Leadership*, 41(8):32–35, 1984.

[Ke14] Fengfeng Ke. An implementation of design-based learning through creating educational computer games: A case study on mathematics learning during design and computing. *Comput. Educ.*, 73:26–39, April 2014.

[KP05a] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137, June 2005.

[KP05b] Claire Kenny and Claus Pahl. Automated tutoring for a database skills training environment. *SIGCSE Bull.*, 37(1):58–62, February 2005.

[KRMA08] Michael J. Kofler, Mark D. Rapport, and R. Matt Alderson. Quantifying adhd classroom inattentiveness, its moderators, and variability: a meta-analytic review. *Journal of Child Psychology and Psychiatry*, 49(1):59–69, 2008.

[KSC$^+$12] Mehdi Kaytoue, Arlei Silva, Loïc Cerf, Wagner Meira, Jr., and Chedy Raïssi. Watch me playing, i am a professional: a first study on video game live streaming. In *Proceedings of the 21st international conference companion on World Wide Web*, WWW '12 Companion, pages 1181–1188, New York, NY, USA, 2012. ACM.

[KW04] Raph Koster and Will Wright. *A Theory of Fun for Game Design*. Paraglyph Press, 2004.

[leg12] Lego group. *LEGO MINDSTORMS Hardware Developer Kit (HDK)*. http://mindstorms.lego.com, August 2012.

[Lew10] Colleen M. Lewis. How programming environment shapes perception, learning and goals: Logo vs. scratch. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 346–350, New York, NY, USA, 2010. ACM.

[LH80] Henry Lieberman and Carl Hewitt. A session with tinker: Interleaving program testing with program design. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages 90–99, New York, NY, USA, 1980. ACM.

[lig12] Armor games. lightbot. http://armorgames.com/play/2205/light-bot, August 2012.

[Lin92] O. R. Lindsley. Why arent effective teaching tools widely adopted? *Journal of Applied Behavior Analysis*, 25:2126, 1992.

[LK11] Michael J. Lee and Andrew J. Ko. Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the seventh international workshop on Computing education research*, ICER '11, pages 109–116, New York, NY, USA, 2011. ACM.

[LKLC11] Conor Linehan, Ben Kirman, Shaun Lawson, and Gail Chan. Practical, appropriate, empirically-validated guidelines for designing educational games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 1979–1988, New York, NY, USA, 2011. ACM.

[LKN99] S.W. Lee, K.E. Kelly, and J.E. Nyre. Preliminary report on the relation of students on-task behavior with completion of school work. *Psychological Reports*, 84:267–272, 1999.

[LL86] J.W. Lloyd and A.B. Loper. Measurement and evaluation of task-related learning behavior: Attention to task and metacognition. *School Psychology Review*, 15(3):336–345, 1986.

[Mac11] Matthew B. MacLaurin. The design of kodu: a tiny visual programming language for children on the xbox 360. *SIGPLAN Not.*, 46(1):241–246, January 2011.

[Mal81a] Thomas W. Malone. Toward a theory of intrinsically motivating instruction. *Cognitive Science*, 5(4):333 – 369, 1981.

[Mal81b] Thomas W. Malone. Toward a theory of intrinsically motivating instruction. *Cognitive Science*, 5(4):333 – 369, 1981.

[May07] Merrilea J. Mayo. Games for science and engineering education. *Commun. ACM*, 50(7):30–35, July 2007.

[mc 15] http://www.gamespot.com/articles/minecraft-passes-100-million-registered-users-14-3-million-sales-on-pc/1100-6417972/, September 2015.

[MdR06] Linda Mannila and Michael de Raadt. An objective comparison of languages for teaching introductory programming. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, Baltic Sea '06, pages 32–37, New York, NY, USA, 2006. ACM.

[MGBMO$^+$08] Pablo Moreno-Ger, Daniel Burgos, Iván Martínez-Ortiz, José Luis Sierra, and Baltasar Fernández-Manjón. Educational game design for online education. *Comput. Hum. Behav.*, 24(6):2530–2540, September 2008.

[ML87] T. W. Malone and M. R. Lepper. Making learning fun: A taxonomy of intrinsic motivations for learning. volume 3, pages 223–253, Hillsdale, N.J., 1987. Erlbaum.

[MM11] Dennis Maciuszek and Alke Martens. A reference architecture for game-based intelligent tutoring. In Patrick Felicia, editor, *Handbook of Research on Improving Learning and Motivation through Educational Games: Multidisciplinary Approaches*, chapter 31, pages 658–682. IGI Global, Hershey, PA, 2011.

[MRR$^+$10] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, November 2010.

[MT08] Laurie Murphy and Lynda Thomas. Dangers of a fixed mindset: implications of self-theories research for computer science education. In *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, ITiCSE '08, pages 271–275, New York, NY, USA, 2008. ACM.

[MTJV09] Mathieu Muratet, Patrice Torguet, Jean-Pierre Jessel, and Fabienne Viallet. Towards a serious game to help students learn computer programming. *Int. J. Comput. Games Technol.*, 2009:3:1–3:12, January 2009.

[MWM12a] Dennis Maciuszek, Martina Weicht, and Alke Martens. Seamless integration of game and learning using modeling and simulation. In *Proceedings of the Winter Simulation Conference*, WSC '12, pages 143:1–143:10. Winter Simulation Conference, 2012.

[MWM12b] Dennis Maciuszek, Martina Weicht, and Alke Martens. Seamless integration of game and learning using modeling and simulation. In *Proceedings of the Winter Simulation Conference*, WSC '12, pages 143:1–143:10. Winter Simulation Conference, 2012.

[NWF$^+$03] Nachiappan Nagappan, Laurie Williams, Miriam Ferzli, Eric Wiebe, Kai Yang, Carol Miller, and Suzanne Balik. Improving the cs1 experience with pair programming. *SIGCSE Bull.*, 35(1):359–362, January 2003.

[OG06] Jackie O'Kelly and J. Paul Gibson. Robocode and problem-based learning: A non-prescriptive approach to teaching programming. *SIGCSE Bull.*, 38(3):217–221, June 2006.

[Ovi06] Sharon Oviatt. Human-centered design meets cognitive load theory: Designing interfaces that help people think. In *Proceedings of the 14th Annual ACM International Conference on Multimedia*, MULTIMEDIA '06, pages 871–880, New York, NY, USA, 2006. ACM.

[OWB05] Harold F. O'Neil, Richard Wainess, and Eva L. Baker. Classification of learning outcomes: evidence from the computer games literature. *The Curriculum Journal*, 16(4):455–474, 2005.

[Pal90] David B. Palumbo. Programming language/problem-solving research: A review of relevant issues. *Review of Educational Research*, 60(1):65–89, 1990.

[Pap80] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA, 1980.

[pap15] http://www.papert.org/articles/doeseasydoit.html, September 2015.

[Pat81] Richard E. Pattis. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1981.

[PCR11] Lakshmi Prayaga, James W. Coffey, and Karen Rasmussen. Strategies to teach game development across age groups. *Int. J. Gaming Comput. Mediat. Simul.*, 3(2):28–43, April 2011.

[Pil03] Nelishia Pillay. Developing intelligent programming tutors for novice programmers. *SIGCSE Bull.*, 35(2):78–82, June 2003.

[Pre01] Marc Prensky. Digital natives, digital immigrants part 2: Do they really think differently? *On the Horizon*, 9(6):1–6, 2001.

[Pre03] Marc Prensky. Digital game-based learning. *Comput. Entertain.*, 1(1):21–21, October 2003.

[Pre04] Marc Prensky. *Digital Game-Based Learning*. McGraw-Hill Pub. Co., 2004.

[Pre05] David Preston. Pair programming as a model of collaborative learning: A review of the research. *J. Comput. Sci. Coll.*, 20(4):39–45, April 2005.

[PSR99] Androniki Panteli, Janet Stack, and Harvie Ramsay. Gender and professional ethics in the it industry. *Journal of Business Ethics*, 22(1):51–61, 1999.

[pur15a] http://www.gamespot.com/forums/xbox-association-1000003/how-often-do-you-buy-games-26403582/, September 2015.

[pur15b] http://www.psu.com/forums/showthread.php/295340-how-often-do-you-buy-video-games, September 2015.

[RCPS01] Eyal M. Reingold, Neil Charness, Marc Pomplun, and Dave M. Stampe. Visual span in expert chess players: Evidence from eye movements. *Psychological Science*, 12:48–55, 2001.

[RD00] Richard M. Ryan and Edward L. Deci. Intrinsic and extrinsic motivations: Classic definitions and new directions. *Contemporary Educational Psychology*, 25(1):54 – 67, 2000.

[Rit09] Albert D. Ritzhaupt. Creating a game development course with limited resources: An evaluation study. *Trans. Comput. Educ.*, 9(1):3:1–3:16, March 2009.

[RRB12] Daniel Roberge, Anthony Rojas, and Ryan Baker. Does the length of time off-task matter? In *Proceedings of the 2Nd International Conference on Learning Analytics and Knowledge*, LAK '12, pages 234–237, New York, NY, USA, 2012. ACM.

[RSL12] Alberto Rizzo, Stefan Schutt, and Dale Linegar. Imagine that: Creating a 'third space' for young people with high functioning autism through the use of technology in a social setting. In *Proceedings of the 24th Australian Computer-Human Interaction Conference*, OzCHI '12, pages 513–516, New York, NY, USA, 2012. ACM.

[RW12] Glen Robertson and Ian Watson. Case-based learning by observation: preliminary work. In *Proceedings of The 8th Australasian Conference on Interactive Entertainment: Playing the System*, IE '12, pages 24:1–24:1, New York, NY, USA, 2012. ACM.

[SA98] Julika Siemer and Marios C. Angelides. Towards an intelligent tutoring system architecture that supports remedial tutoring. *Artif. Intell. Rev.*, 12(6):469–511, 1998.

[Sad10] D. Royce Sadler. Beyond feedback: developing student capability in complex appraisal. *Assessment and Evaluation in Higher Education*, 35(5):535–550, 2010.

[SAMP12] Adam M. Smith, Erik Andersen, Michael Mateas, and Zoran Popović. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, pages 156–163, New York, NY, USA, 2012. ACM.

[SB04] Kurt Squire and Sasha Barab. Replaying history: Engaging urban underserved students in learning world history through computer simulation games. In *Proceedings of the 6th International Conference on Learning Sciences*, ICLS '04, pages 505–512. International Society of the Learning Sciences, 2004.

[SBGH04] Kurt Squire, Mike Barnett, Jamillah M. Grant, and Thomas Higginbotham. Electromagnetism supercharged!: Learning physics with digital simulation games. In *Proceedings of the 6th International Conference on Learning Sciences*, ICLS '04, pages 513–520. International Society of the Learning Sciences, 2004.

[SBS84] Marlene Scardamalia, Carl Bereiter, and Rosanne Steinbach. Teachability of reflective processes in written composition. *Cognitive Science*, 8(2):173–190, 1984.

[SC73a] H. A. Simon and W. G. Chase. Perception in chess. *Cognitive Psychology*, 1:33–81, 1973.

[SC73b] H. A. Simon and W. G. Chase. Skill in chess. *American Scientist*, 61:394–403, 1973.

[SC13] Catherine Schifter and Maria Cipollone. Minecraft as a teaching tool: One case study. In Ron McBride and Michael Searson, editors, *Proceedings of Society for Information Technology and Teacher Education International Conference 2013*, pages 2951–2955, New Orleans, Louisiana, United States, March 2013. AACE.

[SG11] Michael Striewe and Michael Goedicke. Using run time traces in automated programming tutoring. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, pages 303–307, New York, NY, USA, 2011. ACM.

[SGSZ11] Stefan Sobernig, Patrick Gaubatz, Mark Strembeck, and Uwe Zdun. Comparing complexity of api designs: An exploratory experiment on dsl-based framework integration. *SIGPLAN Not.*, 47(3):157–166, October 2011.

[Sha04] David Williamson Shaffer. Pedagogical praxis: The professions as models for post-industrial education. *Teachers College Record*, 106:1401–1421, 2004.

[Sha05] David Williamson Shaffer. Epistemic games. *Journal of Online Education*, 2005.

[Sha15] Adrienne Shaw. *Gaming at the Edge: Sexuality and Gender at the Margins of Gamer Culture*. Univ Of Minnesota Press, 2015.

[SHM$^+$08] Beth Simon, Brian Hanks, Laurie Murphy, Sue Fitzgerald, Renée McCauley, Lynda Thomas, and Carol Zander. Saying isn't necessarily believing: influencing self-theories in computing. In *Proceedings of the Fourth international Workshop on Computing Education Research*, ICER '08, pages 173–184, New York, NY, USA, 2008. ACM.

[SMAoA83] Alan H. Schoenfeld and DC. Mathematical Association of America, Washington. *Problem Solving in the Mathematics Curriculum. A Report, Recommendations, and an Annotated Bibliography. MAA Notes, Number 1 [microform] / Alan H. Schoenfeld.* Distributed by ERIC Clearinghouse, [Washington, D.C.] :, 1983.

[Soh06] Leen-Kiat Soh. Incorporating an intelligent tutoring system into cs1. *SIGCSE Bull.*, 38(1):486–490, March 2006.

[Sol78] N. Solntseff. Programming languages for introductory computing courses: a position paper. *SIGCSE Bull.*, 10(1):119–124, February 1978.

[TG10] Allison Elliott Tew and Mark Guzdial. Developing a validated assessment of fundamental cs1 concepts. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 97–101, New York, NY, USA, 2010. ACM.

[top15] http://www.psu.com/forums/showthread.php/295340-how-often-do-you-buy-video-games, September 2015.

[Tri15] Michael Trice. Putting gamergate in context: How group documentation informs social media activity. In *Proceedings of the 33rd Annual International Conference on the Design of Communication*, SIGDOC '15, pages 37:1–37:5, New York, NY, USA, 2015. ACM.

[TSD$^+$10] Sureyya Tarkan, Vibha Sazawal, Allison Druin, Evan Golub, Elizabeth M. Bonsignore, Greg Walsh, and Zeina Atrash. Toque: designing a cooking-based programming language for and with children.

In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 2417–2426, New York, NY, USA, 2010. ACM.

[vAD04] Luis von Ahn and Laura Dabbish. Labeling images with a computer game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 319–326, New York, NY, USA, 2004. ACM.

[VC78] L. S. Vygotskii and Michael Cole. *Mind in society : the development of higher psychological processes / L. S. Vygotsky ; edited by Michael Cole ... [et al.]*. Harvard University Press, Cambridge :, 1978.

[Wal04] Gary N. Walker. Experimentation in the computer programming lab. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '04, pages 69–72, New York, NY, USA, 2004. ACM.

[WCBn$^+$11] Wayne Ward, Ronald Cole, Daniel Bolaños, Cindy Buchenroth-Martin, Edward Svirsky, Sarel Van Vuuren, Timothy Weston, Jing Zheng, and Lee Becker. My science tutor: A conversational multimedia virtual tutor for elementary school science. *ACM Trans. Speech Lang. Process.*, 7(4):18:1–18:29, August 2011.

[Wen99] Etienne Wenger. *Communities of Practice: Learning, Meaning, and Identity (Learning in Doing: Social, Cognitive and Computational Perspectives)*. Cambridge University Press, 1 edition, September 1999.

[WF98] Barbara Y. White and John R. Frederiksen. Inquiry, Modeling, and Metacognition: Making Science Accessible to All Students. *Cognition and Instruction*, 16(1), 1998.

[Wit53] Ludwig Wittgenstein. *Philosophical Investigations*. Basil Blackwell, Oxford, 1953.

[WKS98] Piotr Winkielman, Brbel Knuper, and Norbert Schwarz. Looking back at anger: Reference periods change the interpretation of emotion frequency questions. In *Journal of Personality and Social Psychology*, volume 75, pages 719–728, Sep 1998.

[wom15] http://www.aauw.org/files/2013/02/graduating-to-a-pay-gap-the-earnings-of-women-and-men-one-year-after-college-graduation.pdf, September 2015.

[WT13] Tianchong Wang and D. Towey. A mobile virtual environment game approach for improving student learning performance in integrated science classes in hong kong international schools. In *Teaching, Assessment and Learning for Engineering (TALE), 2013 IEEE International Conference on*, pages 386–388, Aug 2013.

[YZ09] Wong Seng Yue and Nor Azan Mat Zin. Usability evaluation for history educational games. In *Proceedings of the 2Nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, ICIS '09, pages 1019–1025, New York, NY, USA, 2009. ACM.