Learning Game Design and Software Engineering through a Game Prototyping Experience: A Case Study

Mark Yampolsky[1,2] and Walt Scacchi[2]
[1]Northwood High School
Irvine, CA
and
[2]Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
wscacchi@ics.uci.edu

ABSTRACT
This report describes a case study of small-scale effort in employing game playtesting as a starting point for learning about mainstream issues and challenges found in modern software engineering projects and practices. The goal is to be descriptive and informing through a qualitative rendering, rather than prescriptive and employing a quantitative analysis. This study draws attention to the case of where a student with no prior experience in software development or programming must take on the task of learning how to make a game, and along the way learn about many common challenges in modern SE practice through personal discovery and experience. The game itself also imposes challenges in that we have chosen a new, unfamiliar game genre and domain that emphasizes science learning as its purpose for play. Along the way, we discuss issues in requirements, design, prototyping, testing, user experience assessment, and evolutionary software extension, all prior to a formal education in coding or introductory level Computer Science or SE. Though our efforts may seem unusual or anomalous, we believe our methods are open for adoption and reuse by those interested in lowering the barriers to entry into game software development in specific, and into SE more generally.

Keywords
Game design, game playtesting, science learning games, case study.

1. GAME PLAYTESTING AS AN INTRODUCTION TO COMPUTER GAMES AND SOFTWARE ENGINEERING
Computer games in general, and educational games in particular are often being treated as tools for learning. Along with the entire video game market, the selection of educational games grew, becoming large enough to be considered a genre in the world of games. Unfortunately, games in this genre, especially those that focus on topics in science, technology, engineering or mathematics (STEM) can sometimes have fun take a backseat when it comes to engaging the user, effectively diminishing or eliminating a delivery mechanism that could make even the most mundane material interesting. However, the challenge in STEM game design comes in finding the balance between playful fun and learning through the introduction of engaging game mechanics that playfully immerse the player in the world of a game's particular subject matter.

As part of a preliminary study that analyzed a sample of science-oriented games in order to discover and examine examples of games that succeed in delivering an authentic engagement in science topics [14], a new prototype game, *Beam*, was developed. This effort utilized several practices in game design and software engineering identified within that study. Beam was developed without any prior programming or software development experience on the part of the first

2016 ICSE GAS Workshop, May 16, 2016, Austin, TX, USA.

author, and the study served as his first exposure to a systematic analysis of game design elements and their implementation in a game development environment. This repurposing of gameplay elements is an established concept in game software engineering [20], and we describe how these concepts were surfaced and used throughout the development of the Beam prototype.

This paper presents a case study focusing on the development of the Beam game from a software engineering (SE) perspective. Our purpose is to describe how and what mainstream SE challenges arise during such a development effort, particularly when the game is developed by someone with no prior education or experience in programming or Computer Science (CS), yet who like many students these days, has extensive first-hand experience in playing computer games, as well as general literacy in games. Our goal was not to see if/how such a student might struggle or fail, but instead to focus attention on utilizing an easily replicated approach to introduce students to contemporary challenges in computer games and software engineering (CGSE) [18,20], where tools and techniques can supplement traditional pedagogical schemes. Thus, our efforts follow practices in design science for exploratory information system development [12]. Similarly, our learning goals are qualitative and discovery-oriented, meaning we use this case study to help surface learning experiences in CGSE, rather than to test and refine existing learning models for programming or programming language efficacy [13].

We next briefly describe the genre and domain of computer games that served as our starting point, and then transition into description of a discovery-oriented experience of learning about game design and SE through a game playtesting informed approach. Our choice to examine a new, unfamiliar domain and game genre is to minimize our prior expectations for what to expect or do when developing a game in a familiar domain, or with known design methods and issues. Along the way, we highlight mainstream SE challenges like requirements, design, prototyping, software development environments, and testing, among others. We then provide an reflective assessment of the lessons learned from this case study experience, followed by conclusions that encourage other independent studies and SE Education experiments like this.

2. THE GENRE AND DOMAIN OF SCIENCE LEARNING GAMES

In this study, our goal was to develop a new *Science Learning Game* for learning about optics and beam physics [17]. An SLG is a computer game or game-based simulation whose content, game mechanics, or play experience focus on a domain of scientific research or STEM education [4,10,15,19,20,21]. SLGs are games whose purpose is to help the player learn about authentic problems, techniques, concepts and solutions relevant to a STEM domain, but to go about doing so in playful ways. SLGs are a small and mostly marginalized genre of computer games when one looks at the international computer game industry. However, as some game scholars and educational theorists have observed, many computer games succeed because they are great learning environments that embody both classic and modern theories of constructivist learning, self-identity through role play, reflective thinking, domain-specific specialist language skills, and multi-player socialization. Perhaps such benefits can be realized in SE.

Some SLG's primary purpose is to educate the player on a STEM topic through gameplay. These SLGs are designed to specifically support formal or informal STEM education, where alignment with National Science Education Standards is an important consideration [19,22]. In contrast, other SLGs are focused on engaging players in citizen science, where the core objective is for game players to help solve puzzles or problems that are of interest to scientists working on a complex research challenge that can be factored into pieces addressable through collective or crowdsourced multi-player game play sessions (e.g., [9]). While STEM education may be an indirect goal of such

crowdsourced science games, public engagement in science and assisting in solving outstanding research problems in data analysis are often the primary goals.

Two examples of the genre are *Cellcraft* and *Fold-It* [14]. *Cellcraft* can be classified as a game designed specifically for teaching a subject to students, whereas *Fold-It* is classified as a "citizen science" game, where the primary focus is to use crowdsourcing and scientific concepts to make scientific advancements [9]. *Cellcraft* uses real-time strategy (RTS) game mechanics to teach middle/high school age students the structure and function of a cell. Supplying the player with limited resources to fight off invaders, the game tasks the player with acquiring an understanding of the cell's various parts and how to utilize them to function, requiring the balance of statistics such as amino acids and ATP. *Cellcraft* stood out due to it's repurposing of the common and popular RTS mechanic, yet doing so in a unique way. *Fold-It's* mechanics include basic identification, but include a variety of spatial and planar tools that give a puzzle-like quality. With the point system, accuracy becomes an important attribute to the game. Upon completion, a corrected model constructed with crowd-sourced identification is shown for comparison, to indicate missing or extra components. This process also allows users to learn from their previous game play mistakes and inaccuracies in order to continue to improve their performance scores.

3. PLAYTESTING GAMES AS A BASIS FOR INFORMAL DOMAIN ANALYSIS

3.1 Playtesting as a means of education

Many students have an active interest in playing computer games on a variety of different devices. Some students want to transform their interest in game play to game design or game making. Many games employ game mechanics [1,24] and software development kits (SDKs) that are designed to encourage or embrace game making or modding [18,19,20]. While making and modding are important modalities for learning about game design, they primarily rely on informal game design practices and observational social learning [16] (e.g., via watching online tutorial videos). Such learning modality may be effective, but may also be *ad hoc*, difficult to systematize and assess, and thus inefficient or ineffective. In contrast, formal game development curricula often in the form of undergraduate degree programs have emerged to help provide foundations, principles, and practices for students to learn and engage in game design. The question now is how best to engage students who want to learn about game design, but who may not be positioned to commit to a multi-year educational program, or want to try before they buy into such a program? For example, does computer game development represent an opportune occasion to introduce high school or undergraduate students, or others, to computer science or software engineering? Can students who know little about computer science, but who are familiar with computer game play, social media usage, and Web surfing as their primary modes of exposure to "computational thinking" learn about game design, software engineering, or computer science principles without first learning computer programming? What is an effective way to engage students or others new to game design to learn replicable methods of inquiry and discovery through playtesting games? Questions like these helped motivate this study.

What students can often bring to their first experience in formal game design education is a personal history of games played, as well as a willingness to play new and unfamiliar games. This latter capability can easily align with systematic game playtesting. Playtesting is a common game development method that can be used to evaluate the playability [11] or learning value of a new, unfamiliar game during its development or test marketing. Large game development companies like Microsoft Studios have established a separate applied behavioral research unit that focuses on laboratory-based game playtesting with a curated sample of compensated game play volunteers [6].

Conversely, small independent game developers also seek the benefits of ongoing playtesting within limited budgets [23].

In a previous study, we utilized playtesting of SLGs to support and compare: (a) how to identify and distinguish functional versus non-functional game software requirements; (b) how to playtest complex SLGs; (c) efficacy of human-computer interaction and user experience (UX); and (d) assessment of SLG design, game mechanics, or game play experience [14]. None of these activities specifically requires an introduction to or mastery of computer programming, though programming experience or computational thinking skills may be positive contributing factors. However, there has been little attention to the potential of learning game design or even SE principles from systematic playtesting of a sample of games that are readily amenable to comparative analysis—for example, playtesting a sample of computer games from a single game genre [2,3]. Thus, in our view, playtesting is a core SE activity that supports game software development [8]. Subsequently, we focus on mainstream SE challenges such as requirements, design, software prototyping using an interactive development environment, testing, and user experience assessment, as well as informal domain analysis and modeling, that arose during the development of Beam.

4. PLAYTESTING AND PROTOTYPE-DRIVEN GAME DEVELOPMENT

4.1 Discovering Game Requirements

The evaluation of functional and non-functional game requirements was prevalent from the start of playtesting the SLG sample [14], to the final builds of the Beam game. Perhaps the most rigorous testing focused on how different games ensure that game levels properly balance play challenge and reward. From the initial sample of games, it was clear that a necessary non-functional requirement was to create a working SLG that reacted to the user and had an intuitive, bug-free experience, to allow the player to focus on the engagement and educational values of the SLG play experience. The Beam game aims to teach its players about basic concepts in optics such as: the function of mirrors and lenses; angles of incidence and angles of reflection; refraction; beam splitters; light emitters and detectors; and more, as well as closely related topics in (particle/wave) beam physics [17]. Such topics and associated challenge problems are foundational to understanding more advance topics like photonics, particle accelerator beams, and high fidelity computer graphic ray tracing, as well as quantum teleportation [5].

Some of the gameplay mechanics envisioned for Beam were seen in games such as *Refraction* [cf. 14], and duplicated or repurposed for use in *Beam,* through trial and error during prototyping. The way the game software development kit (SDK) in use, *Construct 2* [7] operates is that it gives the developer generic rules for game play as working code fragments and allows them to be compiled in any order to alter game functionality. Other game SDKs like *GameSalad,* or the open source, *GDevelop*, are similarly based on events and rules, while game SDKs like *Unity* and *Unreal Development Kit* are more complete game (and asset) programming environments. Figure 1 provides a simple screenshot of the *Construct 2* SDK. This encouraged experimentation in the function of these fragments. However, as opposed to the traditional way of teaching coding, prior knowledge of games and that acquired through playtesting gave the first author an advantage in developing the game in that he was aware of what features in previous games he appreciated or disliked, and as such, knew what functionality to include and to avoid in the development of the Beam game.

Through playtesting and experimentation with the tools and online learning media (video tutorials), it was possible to create a learning flow that balanced traditional linear discovery through one's own efforts. Tutorials on the *Construct* online forums, both company created and user generated, provided jumping off points in the development of the code behind *Beam*, however, the game code (event-based rules) was tailored to fit the mechanics of *Beam*. In a sense, development progressed through

what could be deemed as a playtesting of the emerging game prototype via the *Construct 2* SDK the first author was using. Through a process of backup to previous version, experiment, test, fix or revert, then repeat – the student learning this software for the first time was able to find more effective or more efficient ways of completing tasks that were previously completed in an unnecessarily complicated way, such as why and how to use inheritance.

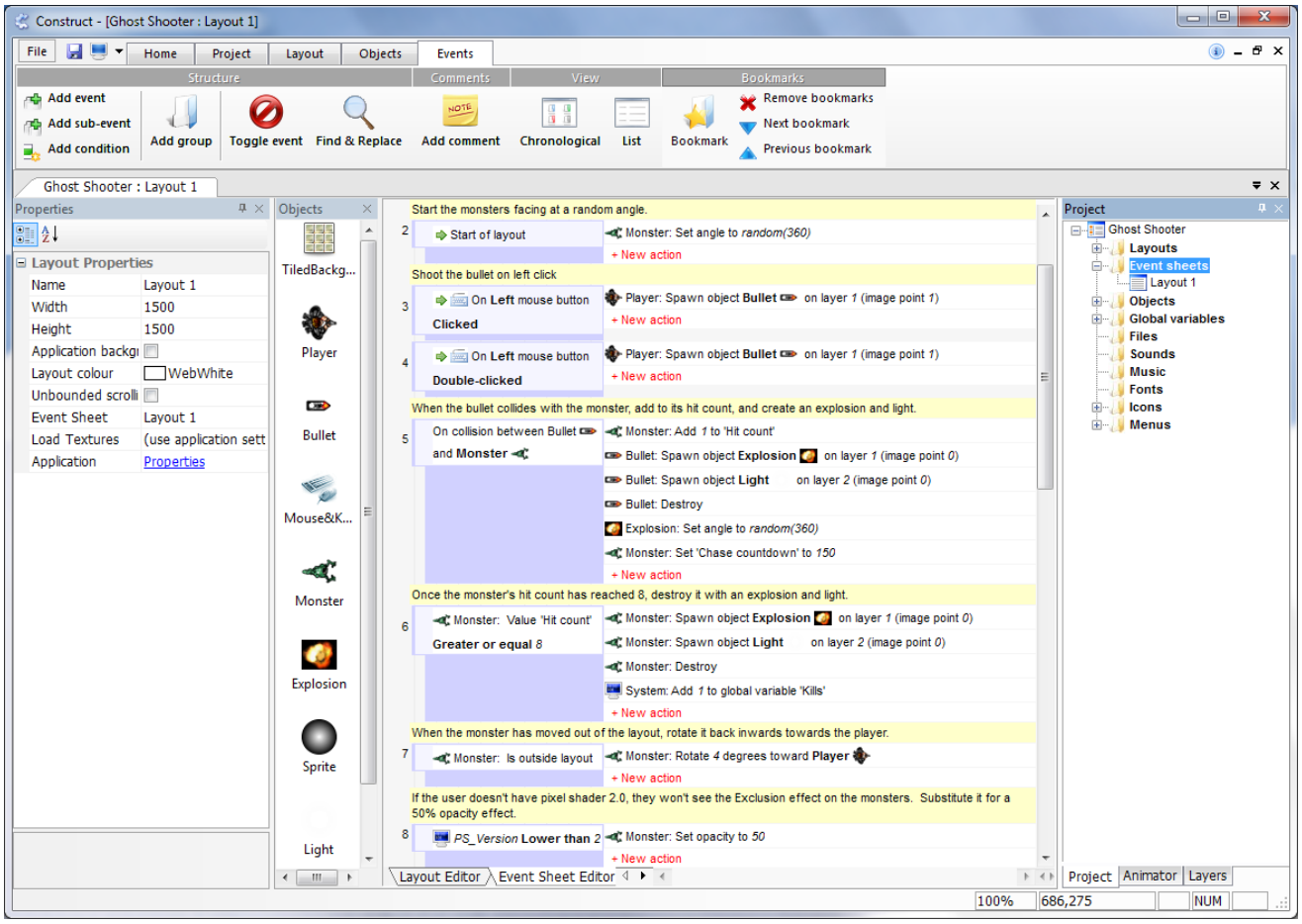

Figure 1. A sample screenshot of the *Construct 2* SDK [7].

This form of playtesting-driven game development also served as a method to discover glitches in the code. *Construct 2* games are event-based and controlled using new/existing rule-sets, as its model of computation for a game/interactive media app. *Construct 2* uses a drag-and-drop development system that prioritizes ease of use and rapid prototyping. What this entails is a series of given "conditions" and "events," the events occurring given the met conditions on a set level layout, each of which can be customized both in a WYSIWYG layout designer and in their code. These can be dragged onto event sheets, eliminating the possibility of syntax errors in their standard format, yet at the same time the software offers a surprising amount of depth in its customization, as it allows the user to break out of its drag-and-drop form and hand write code in Javascript. This implies that general-purpose programming is not readily supported. However, the *Construct 2* export format for games is HTML 5, so that its games can be deployed and run in Web browsers or on mobile devices. So in developing games using *Construct 2*, students do not necessarily learn how to code, but they can make and publish games for the Web and smartphones. So it can serve as an medium for introducing students to learn about CGSE challenges along the way, given some guidance and/or mentoring.

In the given *Construct 2* rule-set, it is possible that some rules work cyclically, activating and deactivating certain functions immediately, hindering certain functions and needlessly taking up computing power. For example, in developing the Beam object that operates a "laser beam" that can course through a Beam optical flow maze, there was a subtle, almost unnoticeable bug where the laser would keep spawning instances even when stopped. This resulted in a gradual lag that would only be noticed after leaving the game running for half an hour. This was only discovered to be the culprit after opening the game in a debug mode and tracking the instances of lasers, allowing the first author another experience for learning about software development through the game making. These sorts of bugs were based on how the rule-set is specified, rather than programming language syntax or typo errors, resulting in situations where it was necessary to fix the scope of control in which the rule-set operated as opposed to the rules themselves. Through exploration of the software, the first author was able to consolidate and refactor the code that had been common in all of the levels, such as starting conditions or the physics behind the lasers, into their own individualized classes, then link the respective levels to the classes they needed.

Overall, as playtesting and prototyping the game progressed, the requirements for Beam became clearer and easier to delineate.

4.2 BEAM Game Design in Hindsight

Given the emerging requirements, the next challenge in CGSE is in articulating the game's overall software system architectural design. As above, the game's design becomes clearest, not at the beginning, but after experience in prototyping and playtesting the emerging Beam game. Overall, the game software is structured by *Constuct 2* to operate as a Web-compatible download (e.g., an HTML 5+Javascript payload) into an end-user's Web browser from a remote Web server (that may or not provide some form of persistent storage, like accumulated user scores). The following figure identifies the functional architecture of the Beam game that emerged through the development effort.

Our purpose in describing the design of Beam is merely to articulate what software functions are supported, and how they are grouped into different rule-sets. As reported elsewhere [19,21], game SDKs like *Construct 2* impose game design choices onto their developers. In this case, all game functions must be codified as event-based rules. This in turn produces a functional rule-set architecture for any game developed within it. This we believe is important to both report and note to students that choice of SE tools can impose their own set of functional requirements and design constraints through their usage. The design of Beam can be summarized as shown in Figure 2.

A. The User Interface includes the main menu and series selection screen the user first sees when entering the game. This are predominantly scripted animations, and also scaling and positioning details of in-game tutorial text, coded within 27 Rules.

B. The Level layouts are a series of level initialization classes that are linked to from the main menu, designed to take the user to the appropriate level, linking to the classes for game mechanics, and coded within 6 Rules.

C. Beam physics contains all of the conditions that determine how the laser beam traverses (routed through) the level based on the positioning of the mirrors and reflection angles, coded within 90 Rules.

C.1 Mirror rotation classes the 360 degree area around the selected mirror into quadrants of variable size to allow for rotation into the 0, 30, 45, 60, and 90 degree intervals, then performs trigonometric calculations to rotate the object to the given quadrant as well as display updating numeric values.

C.2. Mirror zones apply a burn effect onto mirrors located in mirror zones, and also nullify any attempts the game makes to reflect a laser off of the mirror.

D. Scoring Systems keep track of how many times the user tries firing off the laser, and also adds an appropriate amount of points if the user has overlapped the gem object, supplementing these to a score stored in web storage until the series is completed, that are coded in 57 Rules.

E. Startup conditions set preliminary positions for objects such as the laser and hit-scan, as well as initialize all of the web fonts in a given level, coded with 2 Rules.

F. Completion Conditions vary from level to level, but these ensure that the level progresses to the next appropriate level or make sure the user utilize appropriate game mechanics, such as the fiber optic media, and all coded in 18 Rules.

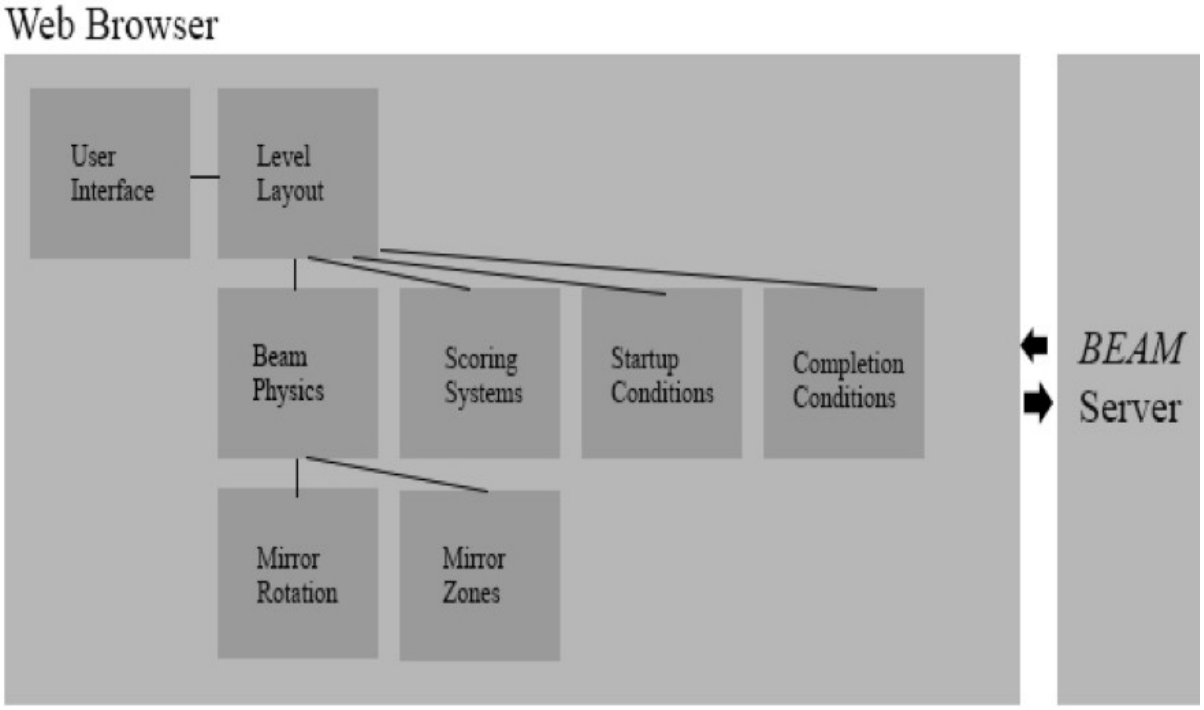

Figure 2. The Beam game's functional architecture: [A] User Interface; [B] Level layout; [C] (Laser) Beam physics; [C.1] mirror rotation; [C.2] mirror zones; [D] Scoring system; [E] startup conditions; [F] end of game completion conditions.

4.3 Implementation: Core Mechanics

Originally, the Beam game prototype lacked an explanatory narrative capability. Such a capability has been partially implemented, while new gameplay extensions to add more game levels, or to vary the gameplay can be introduced in the future. In addition, the game prototyping effort aims to refine the Beam mechanics so that the player explores concepts in optics or beam physics through experimentation. Next, using *Refraction*'s core gameplay as an example, [cf. 14], Beam tasks the player to route laser beams into a beam receiver. In the context of game play task levels in optics, Beam introduced a variety of angles of beam incidence/reflection, these being the 0, 30, 45, 60, and 90 degree intervals of a circle. This was done in order to allow the player to potentially plot out or "design" the optical path of the laser before actually attempting their configuration through mathematics, provided they understood the concept of reflection. In order to discourage the player for

guessing by trying seemingly random layouts, points are subtracted with each firing of the laser. Beam similarly provides a limited number of mirrors to the player in a single level, forcing them to find the best solution with the limited resources. Beam restricts the plane mirrors to three types, ones that can move, rotate, or perform both actions. In addition, in-game "gems" were introduced, to serve the same purpose as achievements in other games, where finding a path to the gem requires the user to beam routes with compound angles.

In an early version of Beam, the hope was to have a consistent laser on the screen that would change in real time as the user altered the mirrors, much like a laser is perceived by our own eyes in an almost instantaneous fashion. Lasers, like all light sources, take time to travel through a space, however the speed at which they travel is so fast we perceive it as instantaneous. This idea was used in the first build of the game to try and develop this real-time laser. When an alteration was made to the play level map, a small, imperceptible sprite is fired out of the laser generator, traveling at a set speed, coming to contact with mirrors and behaving along the laws of reflection or refraction. Whenever the sprite, known as a hit-scan, changed direction, a red laser sprite would be stretched from point A to point B. Originally, this sprite travelled extremely quickly, however, for an unknown reason this caused alignment issues with the laser sprites the user would see unless the sprite was slowed down to a slower pace. At this point, it was easy to see the charting of the laser systems, and with this system a real-time laser would not be possible with the developer's limited technical knowledge, which led to the utilization of some of Construct's proprietary functions, which may have not been programmed in a way that was conducive to accomplish the task at hand in terms of accuracy. The Beam physics rules were then altered so that the laser would always stretch to the position of the hit-scan object, creating the illusion that the laser beam was traveling in slow motion, and when the objects (e.g., mirrors) in the level are moved during play (placement and/or angle of a mirror), the beam resets until triggered again by the player.

Like particulate waves, light particles or wave fronts that make up the laser beam take time to travel from point A to point B, represented respectively in the Beam by the laser beam source and the beam receiver. In the physical world, this takes place at light speeds, but as a result of technical limitations with the SDK utilized, Beam depicts the travel of the light particles at a slow, user-observable pace. Beam used this to its advantage to help the player visualize how light beams travel through the geometry of a complex space, much as is common in educational or research-oriented optical benches. This visualization also allowed for the introduction of a time-related gameplay mechanic because the slowed-down travel in Beam's light travel simulation model was more comprehensible and easily tracked than the near instantaneous movement of light.

4.4 Implementation: Scoring System

Originally, the goal of play in Beam was to reach the end of the level with a certain amount of points. However, there was some leeway given before the point value crosses the threshold in which the player must "Try Again," in order to allow for experimentation on the part of the player. This was also utilized in the higher play levels to increase difficulty by limiting the amount of retries the player had. In the current version of Beam, the scoring system has been significantly restructured to not only accommodate future levels but to ease user navigation through the game as a whole. With the addition of three levels, bringing the total to six, the original UI level selection screen was no longer practical, consisting of a drop down menu on the title screen. The levels are no longer able to be individually selected, rather they are now broken up into two individual "series," each consisting of three levels, which due to length, could be played in succession without the need to return to the menu screen. Rather than fail the player after a given amount of tries, the levels advance regardless of the score of

the player after his/her attempts at the levels, resulting in a cumulative score for the level series. The reasoning behind this decision in scoring was not only make it easier to compare play session performance as a whole, but also to allow the player to master the subject matter throughout the series, giving them the chance to recover points lost in earlier levels. This could be done by intercepting special "gems," often reached through unconventional paths through the level, which would award the player additional points, serving as incentive to boost their score. The cumulative score also brought level unlocks, locking the further series until a sufficient score had been reached that statistically could demonstrate a mastery of the material presented in the series. This also allowed us to increase the difficulty of the gameplay presented in the later series, as well as introduce new gameplay elements, like the fiber spool mechanic.

4.5 Learning from User Experiences

Drawing from several games within the initial playtesting sample, one of the main goals in the development of Beam was to create an intuitive and simple interface system and control scheme. This is part of the reasoning behind the minimalist aesthetic of the game, allowing vibrant color to draw the user's focus to important objects that differentiate themselves from the gray layout, as seen in Figure 3.

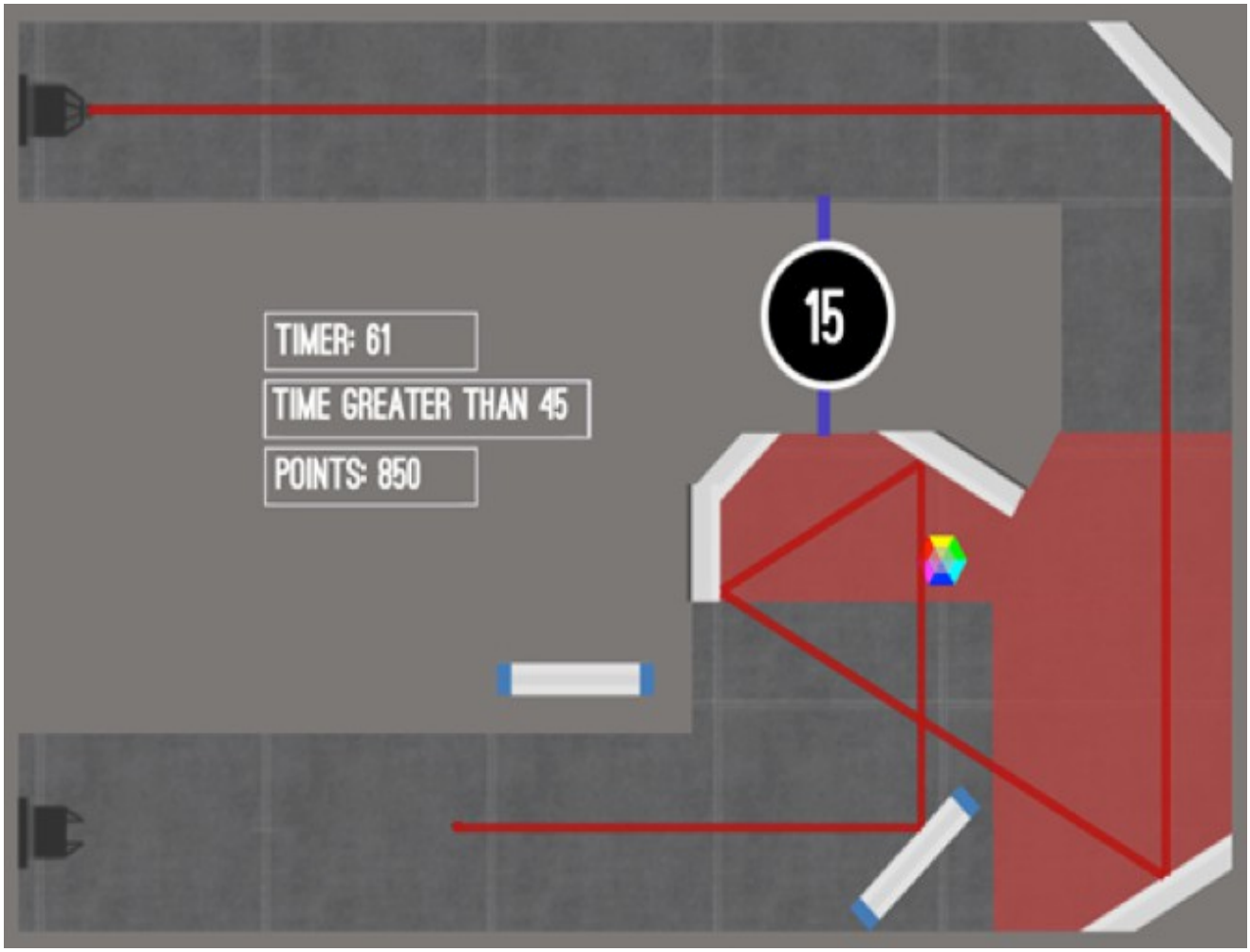

Figure 3. The user interface view of Beam during a play session in a level that utilizes a fiber optic spool ("15"), multiple mirrors for directing or reflecting the laser beam path from that starts from the upper left beam emitter on its path to the lower beam detector.

This is effectively used in multiple instances to indicate the status of objects. The different variations of mirrors are color coded based on their function – red indicating solely movement, green indicating solely rotation, and blue indicating both, whereas stationary mirrors possess only gray color schemes. Coloration is also used with the gem to indicate whether the user has activated, in this case, a Boolean function which grants the user additional points for pursuing an unconventional or more difficult path through the puzzle as noted earlier. This differentiation of color and state allows the user to see potential problems with their solution through visual observation, and then alter their approach seamlessly. To ease user experience, simple indicators were added to buttons and menu icons to show the user that they could be pressed and burns were placed on mirrors when a right click was held to show which had been activated.

Throughout the development process, Beam was playtested extensively by the first author, as well as posted online for playtesting by other volunteer players. As already noted, playtesting serves as a method to make sure that the game like Beam was functioning as intended. Such playtesting was especially important after any significant edit to the game's code, which is of course something all students of software developer need to know and learn through experience. This originally was done in the game's run-time environment and debugging was primarily visual, but as the first author grew accustomed to the software, it became necessary to shift into a debugging mode where it was possible to track all sprite instances in the game, as well as all of the global variables used in making the laser operate. In completed releases, other players provided feedback through playtesting in both functional and non-functional ways. The latter being experiences such as providing feedback on whether playing Beam was "fun" and interesting to unfamiliar users. The former allowed the first author to identify game-breaking issues in the code and remedy them in a timely manner, such as when tested by his high school physics instructor, it was discovered that mirrors meant to be stationary could be moved in a function left activated from the development build, and without this sort of playtesting would have been left as a glitch within the final game likely to be discovered later.

Originally, users were greeted with a start game screen with a dynamic laser that matched the vertical position of their mouse and took the user to a level selection screen where they could choose from three levels provided. Because the levels could be completed in a fairly quick amount of time, the level selection was removed and level selection was integrated into the start game screen, the movement of the laser now showing a tab where the user could select their desired level. This was further improved, as discussed before, with the introduction of multiple game play levels, that resulted in minimizing clutter on the title screen. Level selection buttons in levels were also eliminated to streamline the experience as it was deemed the amount of time it took to complete a level did not warrant a need to return to the main menu screen. In addition, level completion screens were added that darkened and finalized the level layout created by the user, effectively disallowing any movement, and placing a focus on the user's final score. This was later edited with the series scoring system to remove try again screens, as it was deemed that a single run through a series would be a more accurate representation of the user's knowledge.

4.6 Envisioning Evolutionary Extensions to Beam: Accommodating Contemporary Research in Photonics

In an effort to demonstrate if contemporary scientific research issues or problem-solving challenges could be designed into an SLG like Beam, this led us to look for recent advances in beam physics for a suitable problem to address. Said differently, students should be challenged to understand how a given software system or game can be extended to accommodate new functions or extensions, as "modding" is a common software evolutionary design practice in game software design [18,20]. Taking on design of game play mechanics that encode research advances in advanced photonics system design would provide such a goal.

We (second author) found a research paper addressing the design of an experimental quantum teleportation system in *Nature Photonics* [5] that utilizes fiber optic spools to buffer/delay light particle travel over a long distance in a relatively small space. Such spools can confine light beam particles (photons) to a small volume spool space for a given amount of time by simply forcing the light particle to travel the full length of the close looped fiber (physically measured in kilometers [5]), rather than a linear path through the total physical space occupied. Such a mechanic was then added to Beam to allow the player to be given a challenge of designing a beam path limited to time interval within which the laser had to reach the receiver. This allows players to use a combination of the fiber optic loops and speed of light-to-distance time required for the laser beam to traverse the game play challenges in this highest level. Solving challenges in such beam path design and measure of time-to-travel that beam path distance is done by a player in order to satisfy the top level objectives in combination with the points already required. This is facilitated by adding metric indicators to the space, so that along with some basic geometry, players may be able to compute time-to-travel a distance at light speed.

In an anticipated enhancement to Beam, it should be possible to demonstrate the quantum entanglement of two photons traveling on different beam pathways [5]. This idea also entails use of a beam splitter. Common in laser laboratories, adding functionality for a beam splitter would allow new game mechanics, such as the introduction of a new polarized laser, and respectively mirrors that only functioned with the polarized laser beam, as well as completion conditions regarding the two laser types. However, this requires duplicating the entire laser system built in the early Beam version, side by side with the core laser, until some further refactoring of the game's functionality can be put into place.

5. LESSONS LEARNED

Five lessons that follow from this case study and our experience along the way may be of value to other software engineering researchers and educators interested in computer game development can be identified as follows:

1. It is possible to enable students who are already accomplished game players, as well as game literate, to learn how to make operational software prototypes of new games within a domain of study prior to a formal introductory education in computer science or software engineering. This means we can lower the barrier for early entry and exposure to SE practices and principles, and to making operational software systems, when supported with existing game software development kits (SDKs).

2. Such experience can serve to identify and introduce students to core challenges in SE such as requirements, software system design, prototyping, testing, and user experience assessment, both before and during their efforts to learn programming and computational thinking. However, these experiences are intended to be introductory rather than definitive, and they do entail constraints that are imposed by the choice of game SDKs employed [21]. But with pedagogical, instructional, or tutorial guidance, students new to software development, CS or SE can learn through experience and personal discovery some of the core challenges of modern SE practices, at the beginning or preceding their formal education in CS or SE coursework.

3. Simple 2D game SDKs exist in accessible, low-cost forms that allow students or other end-users not skilled in game development or SE to acquire first-hand experience and awareness of core SE challenges, as well as exposure/preview of what lies ahead if they want to pursue studies in computer games and software engineering [cf. 8]. From an educational or learning science perspective, these SDKs provide a scaffolded learning platform for computational experimentation with operational game mechanisms and powerful software development constructs like events and rule-based control schemes [19]. As these SDKs generally provide a single model of computation for game design—e.g.,

an event-driven rule-based system—then students can also be exposed to the power of such domain-specific models as the basis for understanding and designing operational software systems that are compatible with the Web and mobile devices.

4. Playtesting of computer games within a single domain or genre can be a systematic approach to learning about game design practices and software functionality choices via comparative testing of operational games and user experience (with game mechanics as software-based functions), which are contemporary challenges in CGSE [8,21].

5. Through domain-specific game playtesting [14], students can begin to experience advanced SE techniques like domain analysis rubrics, and through prototyping, domain modeling, all before learning CS approaches to programming.

While these lessons may seem obvious in hindsight, they also serve as guidance and encouragement to recognize that innovations in SE Education, CS and computational thinking do not need to start from a traditional game coding or programming perspective [13].

Commonly, it is assumed that students must first study programming language syntax and constructs, control flow, procedure/method invocation, and data structures with trivial or obscure software applications (e.g., "Hello World" programs). Instead, the lessons identified and the experiences reported in this paper suggest that we can now leverage the fact that large populations of students have much first-hand knowledge and experience in using complex, yet entertaining software systems that denote modern computer games. It is thus incumbent on us to consider new ways and means to engage their learning and knowledge discovery through computational media they may already experience and use across multiple system instances. Similarly, "coding first" approaches to CS or SE may not be the only, nor even the most effective, way to engage students who have acquired some literacy with complex software applications that deliver diverse user experiences of varying quality across systems, as well as within or across game subject domains.

This paper thus serves to point to another direction for exploration and advancement within and through the domain of computer games and software engineering.

6. CONCLUSIONS

In this paper, we presented a case study that accounts for the development of an operational game prototype that was informed by prior game playtesting. This study provides a small-scale examination of challenges that can be discovered and experienced by students new to computer game design, Computer Science and Software Engineering. Unsurprisingly, it is not much of a challenge to assign students to play games as part of their study/coursework, and find their response eager and enthusiastic. Systematic game playtesting just adds structure and analytical rubrics to their efforts, as well as providing a basis for comparative analysis, both by individual students as well as across groups of students [14]. This means we can begin to expose students very early in their higher education to contemporary challenges found in the domain of computer games and software engineering very early in their formal academic studies.

This case study examines and describes the development of a prototype science learning game by a student new to CS and SE. Along the way, we show that such a learning experience—the process and practice of learning how to make a unfamiliar computer game in a new domain—surfaces many classic issues and challenges in SE. But we hold that introducing students to these learning experiences is both fun and constructive, as well as potentially transformative for the participating students. We therefore encourage other scholars to try similar undertakings.

7. ACKNOWLEDGMENTS

The research described in this report was supported by grants #1256593 from the National Science Foundation. No endorsement, review, or approval implied. Isabella Villano, San Joaquin High School, provided helpful comments on this paper. Last, the first author, Mark Yampolsky is now a freshman majoring in Computer Science and Game Design at the University of Southern California.

8. REFERENCES

[1] Adams, E. and Dormans, J. (2012). *Game Mechanics: Advanced Game Design*, New Riders, Berkeley, CA.

[2] Apperley, T. (2006). Genre and game studies: Toward a critical approach to video game genres. *Simulation & Gaming*, 37(1), 6-23.

[3] Arsenault, D. (2009). Video Game Genre, Evolution and Innovation, *Eludamos J. Computer Game Culture*, 3(2).

[4] Beier, M.E. Miller, L.M and Wang, S. (2012). Science Games and the Development of Scientific Possible Selves. *Cultural Studies of Science Education*, 7, 963-978.

[5] Bussières, F. Clausen, C. Tiranov, A. *et al.* (2014). Quantum teleportation from a telecom-wavelength photon to a solid-state quantum memory, *Nature Photonics*, 8, 775–778. doi:10.1038/nphoton.2014.215

[6] Butterworth, S. (2014). The Xbox Science Machine, *Official Xbox Magazine*, May 2014.

[7] *Construct 2 Javascript SDK Documentation*, Scirra.com, https://www.scirra.com/manual/15/sdk, accessed 21 January 2016.

[8] Cooper, K. and Scacchi, W. (Eds.) (2015). *Computer Games and Software Engineering*, CRC Press, Taylor & Francis Inc., Boca Raton, FL.

[9] Cooper, S. Khatib, F. Treuille, A. *et al*. (2010). Predicting Protein Structure with a Multiplayer Game, *Nature*, 466, 756–760, (2010).

[10] Dodo, A. Cicchirillo, V. Atkinson, L. and Marx, S. (2014). Portrayals of Technoscience in Video Games: A Potential Avenue for Informal Science Learning. *Science Communication*, 36(2), 219-247.

[11] Desurvire, H. Caplan, M and Toth, J.A. (2004). Using Heuristics to Evaluate the Playability of Games, *Proc. 2004 Computer-Human Interaction Conf.* (CHI 2004), Vienna.

[12] Hevner, A.R. March, S.T. Jinsoo Park, and Ram S. (2004). Design Science in Information Systems Research, *MIS Quarterly*, 28(1), 75-105.

[13] Kelleher, C. and Pausch, R. (2005). Lowering the Barriers to Programming: A Survey of Programming Environments and Languages for Novice Programmers, *ACM Computing Surveys*, 37(2), 83-137.

[14] Lim, R. Yampolsky, M. and Scacchi, W. (2014). *Making Learning Fun: An Analysis of Game Design in Science Learning Games*, ISR Technical Report, UCI-ISR-14-3, (74 pages), October 2014.

[15] Morris, B.J. Croker, S. Zimmerman, C. Gill, D. and Romig, C. (2013). Gaming Science: The "Gamification" of Scientific Thinking, *Frontiers in Psychology*, 4 (Article 607). doi: 10.3389/fpsyg.2013.00607

[16] *Observational Learning,* http://en.wikipedia.org/wiki/Observational_learning, accessed 5 January 2016.

[17] Rosenzweig, J.B. (2003). *Fundamentals of Beam Physics*. Oxford University Press, Oxford, UK.

[18] Scacchi, W. (2010). Computer Game Mods, Modders, Modding, and the Mod Scene, *First Monday*, 15(5), 2010.

[19] Scacchi, W. (Ed.) (2012). *The Future of Research in Computer Games and Virtual Worlds: NSF Workshop Report*, Technical Report UCI-ISR-12-8, Institute for Software Research, University of California, Irvine.

[20] Scacchi, W. (2015). Repurposing Game Play Mechanics as a Technique for Designing Game-based Virtual Worlds, in Cooper, K. and Scacchi, W. (Eds.) (2015). *Computer Games and Software Engineering*, 241-259, CRC Press, Taylor & Francis Inc., Boca Raton, FL.

[21] Scacchi, W. and Cooper, K.M. (2015). Research Challenges at the Intersection of Computer Games and Software Engineering, *Proc. 2015 Conf. Foundations of Digital Games* (FDG 2015), Pacific Grove, CA, June 2015.

[22] Scacchi, W. Nideffer, R. and Adams, J. (2008). Collaborative Game Environments for Informal Science Education: DinoQuest and DinoQuest Online, *Proc. IEEE Conf. Collaboration Technology and Systems* (CTS 2008), Irvine, CA 229-236.

[23] Schanuel, I. (2014). User Testing with Limited Resources, *Gamasutra.com*, accessed 6 September 2014.

[24] Sicart, M. (2008). Defining Game Mechanics, *Game Studies*, 8(2), December.