Building Casual Games and APIs for Teaching Introductory Programming Concepts

Brian Chau
Comp. & Software Sys.
U of Washington Bothell
chautime@msn.com

Rob Nash
Comp. & Software Sys.
U of Washington Bothell
rynn@uw.edu

Kelvin Sung
Comp. & Software Sys.
U of Washington Bothell
ksung@uw.edu

Jason Pace
Digital Future Lab
U of Washington Bothell
jasonpa@uw.edu

ABSTRACT
We are building a series of custom casual games to support teaching and learning of introductory programming (CS1/2) concepts with a focus on ease of adoption. Our games are innovative twists on popular casual genres, and each game is designed explicitly for teaching specific programming concepts (e.g., conditionals, arrays). Based on these games, faculty can explain and students can explore CS1/2 concepts through engaging gameplay mechanics by working with a simple Application Programming Interface (API) defined for each game. Faculty can construct small and fun games to demonstrate concepts while students can exercise their own understanding and creativity by customizing the game and making it their own. To verify the effectiveness and to ensure educational objectives can be accomplished, sample teaching materials have been developed using these APIs. To support selective adoption of the materials by faculty, the games are well-encapsulated and completely independent from one another. To ensure fun and engaging experiences for students, each game is designed, built, and play tested almost entirely by undergraduate students. Based on two completed games and the associated teaching materials, feedback from novice student programmers indicates that the games are engaging and the associated APIs are straightforward to use. This paper presents our motivation and process for building casual games, and discusses the API development and results.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education.*

General Terms
Design, Experimentation

Keywords
Computer Science Education, API, API Design, Casual Games, CS1/2, Programming

1. INTRODUCTION
When properly integrated into coursework, using videogames to teach computer science (CS) accomplishes desired student learning outcomes, builds excitement and enthusiasm for the discipline, and attracts a bright new generation of students early in their academic careers [1], [2]. As a relatively new approach, interested faculty require assistance in the form of elementary sample materials and tutorials to support their exploration and experimentation [3].

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

The Game-Themed Computer Science (GTCS) project and the associated library [3] are designed specifically for this purpose with elaborate sets of sample teaching materials that hide the graphics and gaming details [4]. The self-contained nature of the materials allow faculty to adopt and use each without significant modification to their existing classes. Results from adopting GTCS materials have demonstrated effectiveness in engaging students and achieving the desired learning outcomes [5]–[7]. In addition, results from the many workshops (e.g., [8], [9]) showed that although interested faculty members with no background in graphics or gaming found the GTCS materials to be nontrivial, they were able to comprehend and begin developing game applications based on the GTCS library within a matter of hours [3].

Student feedback on GTCS materials indicated that though they find the materials motivating, they were also frustrated by the simplicity, e.g., the absence of fundamental gaming features like power-ups or win conditions. On the other hand, faculty workshop participants pointed out that the most demanding efforts in building game-specific teaching materials are often unrelated to the educational goals which include the time-intensive processes of locating or generating art and audio assets, or implementing the annoying details of various object interaction rules.

To address this seemingly contradictory feedback while preserving the important characteristics of simplicity and usability for a targeted curriculum, the GTCS project group is building a series of causal games and corresponding APIs. These standalone games each showcase one or two programming concepts, allowing faculty to pick and choose for selective adoption. The games have gone through elaborate playtesting to ensure an engaging and complete gameplay experience. Each API is methodically extracted from the finished game and refined based on usability and support for the presentation of targeted programming concepts so that faculty can build their own aesthetically engaging materials while focusing on the pedagogy rather than irrelevant details such as asset management.

Currently, there are five games under development in various stages of completion. Two of the games in particular—Space Smasher and Corrupted—include finalized APIs, and the sample teaching materials for Space Smasher are currently being field tested in CS1/2 classrooms. This paper uses these two games and their respective APIs as examples to discuss our game development and API refinement processes and results.

In the rest of this paper, section 2 briefly surveys previous work; section 3 reviews existing API studies and articulates a design guideline for our game APIs; section 4 discusses our game and API development processes; section 5 presents our APIs from Space Smasher and Corrupted; and section 6 concludes the paper.

2. GAMES AND CS1/2 CLASSES
Existing work on presenting CS1/2 concepts in the context of computer games can be broadly categorized into three approaches [4]: little or no game programming (e.g., [10]) where students learn by *playing* custom games; per-assignment game

development (e.g., [11]) where individual assignments are games designed around technical topics being studied, and extensive game development where faculty and students work with custom game engines (e.g., [12]) or specialized environments (e.g., [13]).

As pointed out by Levy and Ben-Ari [14] and Ni [15], issues that faculty consider when examining new and innovative teaching materials for adoption include (amongst others): preparation time, material contents, and topic coverage. Yet, most of the existing results from integrating games in CS1/2 classes are typically from faculty members with expertise in graphics or games and are "*student-centric*," where the main goals are student engagement and various learning outcomes—preparation time for adoption and flexibility of the materials for topic coverage are usually not primary concerns. Indeed, it can be challenging to take advantage of these results for the general faculty teaching CS1/2, since many have little to no background in computer graphics or games.

As discussed in the previous section, while effective in addressing the issues of faculty background and curricula modularity, the feedback from previous GTCS materials identified the seemingly contradictory student desire for complexity and faculty need for simplicity [3]. This paper presents the recent GTCS project group efforts in addressing this interesting dichotomy—building complete casual games that offer meaningful gameplay experiences for the students while designing APIs that ensure simple and straightforward curriculum development for the faculty.

3. API DESIGN GUIDELINES

An API can be described as a well-defined interface that exposes the external services of a singular component to clients who will consume these services as elemental software building blocks [16]. In our case of designing an interface for developing casual games, our APIs are a collection of functions intended to be reused by other programmers to perform common tasks that may be difficult, cumbersome, or tedious [17].

Well-defined APIs foster productivity, code reuse, encapsulation of complex systems, and consistent behaviors for their users [18]. A sound API should be easy to use and hard to misuse [19]. In the design of our API, some of the fundamental goals overlap (e.g. productivity and code reuse) while others oppose one another, offering interesting challenges. For example, while achieving tight encapsulation is desirable, our APIs must purposefully expose details relating to the concepts that they are designed to teach, e.g., exposing the details of underlying 2D arrays that represent the grid system of Corrupted for student manipulation. To add to this, our end users are on the two ends of the spectrum of programming expertise—faculty and students of CS1/2 classes.

To address our requirements, the following considerations are articulated to guide the design of our APIs.

- **Usability & Structured Simplicity**: usability and simplicity facilitate the creation of effective CS1/2 materials accessible to students and faculty with no background in graphics or gaming.
- **Discoverability & Learnability**: our end users must be able to build simple applications quickly with minimum familiarity of the APIs but also have the opportunity to gradually explore advanced functionality at their own pace.
- **Expressiveness & Productivity**: While the final products built by the students may resemble casual games, the APIs primarily support the building of effective teaching materials as vehicles for educational content delivery.
- **Encapsulation & Modularity**: as previously discussed, the goals are to tightly encapsulate the complex graphics and gaming functionality while strategically exposing selected concepts for teaching purposes.

Though important, performance is only a referencing factor in our API design—as long as an acceptable frame rate and memory footprint are maintained.

4. THE DEVELOPMENT PROCESSES

The ideal game design and implementation for our purposes must be simple enough so that neither students nor faculty become mired in graphics or game complexity, and yet the gameplay must be genuinely fun so that students can connect their work to a final experience that stands on its own merits. Fortunately, simple and fun are not mutually exclusive when it comes to game design.

4.1 The Two Games

Space Smasher is a variant of Super Breakout-style games, where players remove blocks on the screen by bouncing a ball with a moveable paddle. The popularity of this type of game spans generations, and the numerous variants tend to be largely identical and feature basic color block graphics. Space Smasher introduces more interesting gameplay by adding *customizable* blocks that are capable of triggering events such as swapping blocks, or enabling/disabling unbreakable blocks. Also included are more premium sets of graphics tiles and sounds than are typically found in this genre. The ball-block-paddle collision tests, the special event logic, and the iteration through all blocks present an excellent structured sandbox for teaching and playing with conditionals and loops conceptually.

Corrupted is a variation on the Bubble Shooter genre, where players launch a colored tile into a larger group of tiles and attempt to make matches of three until all tiles are removed or until the tiles advance to meet the player at the opposite end of the screen [20]. Bubble shooters also tend to be simple match-and-remove games with minimalist graphics. Corrupted recasts the game with an active automated opponent employing a variety of tricks to increase the challenge and intrigue, and the game itself has been given a distinct artistic style. The visual, spatial, and multi-dimensional aspects of the color tiles present a rich domain of concepts for use in teaching 1D and 2D arrays.

Note that it is relatively straightforward to design custom levels and additional gameplay elements to both Space Smasher and Corrupted. The game mechanics for either do not require extensive balancing or tuning to make levels enjoyable. Even novice designers can quickly create fun and challenging custom levels. In this way, after a basic process of discovery and familiarization with the concepts and APIs, students can implement their own unique levels as practice exercises.

4.2 The Teams

We knew from the outset that creating casual games designed both to teach fundamental programming concepts and to engage players would require teams with varying expertise: making fun games is an interdisciplinary undertaking and requires a wide variety of skills. Thus, we established a partnership that includes frontline CS1/2 faculty members and the Digital Future Lab (DFL)[1] of the University of Washington Bothell (UWB) campus—an interactive media R&D studio developing original interactive works supporting education, entertainment, and social justice.

The DFL emphasizes students' dual role as creators and learners, while working with students across disciplines and backgrounds to ensure each core component of the game is well designed. The games are developed almost entirely by undergraduate students, including CS and non-CS majors. Roles

[1] http://www.bothell.washington.edu/digitalfuture

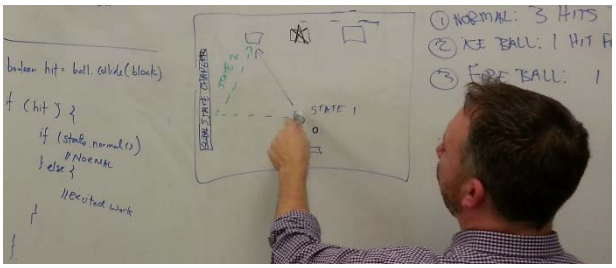


Figure 1: Initial whiteboard sketches for Space Smasher.

include level design, visual and user-interface design, sound and music composition, game development, testing, and project management.

4.3 Game Development Process

Our design approach begins with a unique brainstorming process where game designers generate ideas using familiar game mechanics to facilitate learning while practicing a specific programming concept, and also to gauge the fun factor of the prescribed activities. Faculty contributors help guide the design towards modular and feasible outcomes by providing simple and clearly defined coding requirements.

The photo in Figure 1 exemplifies our unique approach with the initial whiteboard sketches for the Space Smasher game. Notice that the left side of the board charts the "if" control structure that the game is being designed to teach, the center shows a gameplay screen mockup, and basic game rules are listed on the right. The conditional programming construct (on the left) provides the underlying impulse guiding the design process from its initial stages.

As an initial idea gains momentum, the DFL designers examine and refine fundamental gaming mechanics to maximize overall entertainment value, while CS1/2 faculty evaluate implementation simplicity and the level of exposure to the intended programming concept. Simultaneously, student developers create simple digital prototypes so designers can experiment interactively with their ideas and make improvements to the core gaming experience. Playable versions of the game then undergo rigorous hands-on testing to refine each design choice or mechanic.

4.4 API Definition and Refinement Process

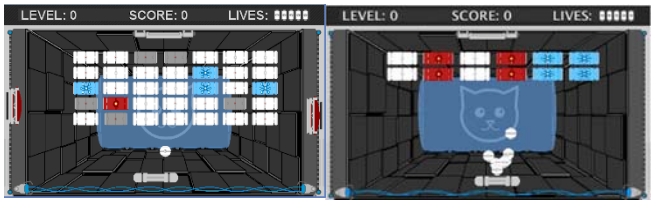
The final game and corresponding API development overlap significantly through a three-step process: (1) finalize the game prototype, (2) define and refine the API while completing the game implementation, and (3) build tutorials and teaching materials based on the API. In our process, the API refinement spans from the stabilization of the prototype game until the team finishes tutorials and teaching materials. This allows for verification of initial usability of the APIs [21].

Once these pieces are in place, the team fine-tunes gameplay while integrating production-quality graphic and audio assets. The result is a game that has the look-and-feel of a studio-quality production, while containing library and game features that will challenge new CS students to program individual game variations as part of a larger learning process.

5. RESULTS

The APIs are defined based on our previous experience from the GTCS foundations game engine [6] where the user code subclasses from an API-defined superclass and overrides two protected methods: *initialize()* and *update()*. The API calls the *initialize()* method exactly once before the game begins and the *update()* method continuously at a real-time rate until the game ends.

The underlying philosophy of the API is to provide all the functionality such that user code can focus on implementing just the




```
public class MySpaceSmasherGame extends SpaceSmasher {
  // init the game, called once by the API
  protected void initialize() {
     lifeSet.add(5);                  //create 5 lives: show in top-left
     paddleSet.add(1);                //create 1 paddle
     blockSet.setBlocksPerRow(6);     //num of blocks per row
     for (int i = 0; i < 2; i++) {    //create two rows of blocks
        blockSet.addNormalBlock(1);      //normal block (light gray)
        blockSet.addFireBlock(1);        //fire block (red)
        blockSet.addNormalBlock(1);      //normal block (light gray)
        blockSet.addFireBlock(1);        //fire block (red)
        blockSet.addFreezingBlock(1);    //ice block (blue)
        blockSet.addFreezingBlock(1);    //ice block (blue)
     }
  }

  //update is called continuously >40 times per second
  protected void update() {
     //control the paddle left/right movement
     Paddle paddle = paddleSet.get(0);    //get the paddle
     if (keyboard.isButtonDown(KeyEvent.VK_LEFT))
        paddle.moveLeft();                //move paddle left
     if (keyboard.isButtonDown(KeyEvent.VK_RIGHT))
        paddle.moveRight();               //move paddle right

     //conditionally spawning balls with loops
     if (keyboard.isButtonDown(KeyEvent.VK_1)) {
        Ball foo = new Ball();         //make a new ball
        ballSet.add(foo);              //add to set of balls
        foo.spawn(paddle);             //put it on screen above the paddle
     } else if (keyboard.isButtonDown(KeyEvent.VK_2)) {
        for (int i=0; i<2; i++) {      //do the logic in loop twice
           Ball foo = new Ball();         //make a new ball
           ballSet.add(foo);              //add to set of balls
           foo.spawn(paddle);             //put it on screen above the paddle
        }
     } else if (keyboard.isButtonDown(KeyEvent.VK_3)) {
        for (int i=0; i<3; i++) {      //do the logic in loop thrice
           Ball foo = new Ball();         //make a new ball
           ballSet.add(foo);              //add to set of balls
           foo.spawn(paddle);             //put it on screen above the paddle
        }
     }
   }
}
```

Figure 2: Space Smasher and Conditional/Loop Example

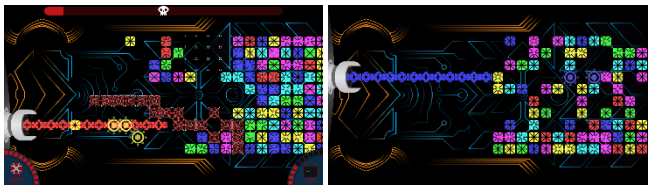
game logic that targets the selected CS1/2 concepts. This adds responsibilities to the API as it must anticipate and provide a slew of resources to accomplish such as exercises, including pre-defined win & lose menu screens, access to all art and audio assets (user code can override these if desired), and anything related to the gameplay environment (e.g., window size, UI layout, etc.). In addition, the APIs provide access to each in-game object (e.g., balls, paddles, or tiles) and their behaviors (e.g., move, speed, remove) plus all potential object-to-object interactions (e.g., ball-and-block collisions, reflections). All created objects are drawn automatically unless explicitly removed or set to invisible. To ensure user testability, mouse and keyboard input are both supported and special debugging modes are built into each API (e.g., stopping and allowing the player to control the ball movement in Space Smasher, or key binding to create specific colored tiles in Corrupted).

In the following we present actual sample teaching materials for Space Smasher and Corrupted to further illustrate each API.

5.1 The Space Smasher API

The screen shots (left: full game, right: teaching example) and code listing in Figure 2 show an example for teaching conditionals and loops. Note that the *MySpaceSmasherGame* class is a subclass of the API defined *SpaceSmasher* superclass. The entire game is then defined by the two protected overridden methods: *inintialize()* and *update()*. Note that game object sets (e.g., lifeSet, paddleSet) are pre-defined with intuitive and convenient behaviors (e.g., add, get, moveLeft, moveRight, etc.).

In this example, the *initialize()* method creates five lives, a paddle, and uses a “for” loop to generate the two rows of various blocks. The *update()* method showcases the simple and chained conditional statements to parse user input and the corresponding



```
public class ColorLineBlaster extends Corrupted {
    private int width = 26;               //2D array width
    private int height = 10;              //2D array height
    private GridElement[][] myTileGrid;   //the 2D array grid

    //init the game, called once by the API
    protected void initialize() {
        myTileGrid = new GridElement[width][height];  //array allocation
        for (int x = width/2; x < width; x++)         //array iteration
            for(int y = 0; y < height; y++)
                myTileGrid[x][y] = new Tile(this);     //create a random color tile
        setTileGrid(myTileGrid);                       //pass the array to the API
    }

    //update is called is called continuously >40 times per second
    protected void update() {
        //player (vertical) position control
        if(keyboard.isButtonTapped(KeyEvent.VK_UP))
            movePlayerUp();   //up moves the player up
        else if(keyboard.isButtonTapped(KeyEvent.VK_DOWN))
            movePlayerDown();// down moves the player down

        //right arrow destroys tiles on the player's height that matches
        // the color of the player then sets the player's color to a random color.
        else if(keyboard.isButtonTapped(KeyEvent.VK_RIGHT)){
            GridElement.ColorEnum currentColor = getPlayerColorEnum();
            int y = getPlayerHeight();
            for (int x = 0; x < getWidth(); x++){
                if (myTileGrid[x][y] != null)
                    if(myTileGrid[x][y].getColorEnum().equals(currentColor)) {
                        myTileGrid[x][y].markForDelete();      //delete the tile
                        myTileGrid[x][y] = null;
                    }
            }
            setPlayerColorEnum(tileHelper.getRandomExistingColor());
        }
    }
}
```

Figure 3: Corrupted and Array Example

responses with the simple "for" loops. The right screen shot at the top of Figure 2 captures the program after the player types in a series of 1, 2, and 3's.

In this case, since the student's code did not define the gaming logic necessary to support multiple balls with collision detection, all the spawned balls will travel through the blocks and window bounds and disappear. After leading students through initial interaction with this example, it is an excellent opportunity to introduce the ball *collide()* and *reflect()* methods and engage students in articulating solutions to keep the balls within the game window bounds and clearing all the blocks.

5.2 The Corrupted API

The screen shots (left: full game, right: teaching example) and code listing in Figure 3 show games developed based on the Corrupted API for teaching 2D arrays. In the *ColorLineBlaster* code, notice the similarity in structure to the Space Smasher example in Figure 2, where the subclass from the API-defined *Corrupted* class overrides the *initialize()* and *update()* methods. Once again, we require from the API intuitive and convenient pre-defined behaviors (e.g., player movements, tile color access).

In this example, the *initialize()* method populates a 2D grid array with random color tiles. The *update()* method polls for keyboard input and triggers game behavior correspondingly. The up and down keys move the player's cannon and the right key clears all tiles in a given row that match the color of the player's cannon. This example highlights linear searching by traversing an array and checking for matching colors as novel animations illustrate the results of their code graphically onscreen.

5.3 Discussions

As exemplified in the case studies of Space Smasher and Corrupted, our casual games are rich yet simple platforms for prominently showcasing CS1/2 programming constructs with support to build engaging user interaction. The APIs are bridges between classic textbook teaching examples and game-themed teaching materials. This unity is achieved by exposing only the familiar programming constructs used in a typical CS 1/2 course while hiding the details that govern image rendering, sound and animation, which may be foreign to students and faculty.

When evaluated against our own design guidelines, the structure of both APIs is based on simple subclass extensions and requires only two functions to implement. With the provided sample teaching materials, a faculty member can begin experimentation without referencing the API documentation. Each casual game and corresponding game objects (e.g., balls, paddles, or grid cells) provide an excellent digital playground for discoveries—users can experience functionality by interacting with the provided games and explore the defined objects to learn more about the API. With the game design being driven by programming concepts, as illustrated in the listings of Figures 2 and 3 it is straightforward to build examples showcasing desired educational concepts. Lastly, we have hidden all art and audio assets, as well as the implementation of game rules, allowing faculty and students to focus on the more important programming constructs at hand.

As an educational tool, the APIs provide students with a sandbox framework that allows them to build simple games and applications. By providing rich visualizations and supporting engaged interactions, students can develop creative games that capture their interest while they learn and explore concepts. A full set of short tutorials and sample materials similar to those of Figures 2 and 3 are freely available on our project website.[2]

6. CONCLUSION

While welcomed by both students and faculty, adopting examples of game-themed materials from the earlier GTCS efforts presented an interesting dichotomy—students requested a more complete and sophisticated gaming experience while faculty demanded less effort in composing art assets and handling game rules when building the teaching materials. The GTCS project team responded to this seemingly contradictory feedback by building causal games and then designing practical APIs to support these games. In this way, students can interact with the games first before delving into the challenging concepts and faculty can construct rich and engaging examples that showcase their selected educational concepts with only a small amount of code and time invested.

Of the five games in development, Space Smasher and Corrupted are the most complete and include refined APIs. The sample teaching materials based on Space Smasher focus on conditionals and loops, and are currently being field tested by faculty members with no background in graphics or games with encouraging preliminary feedback.

The appeal of video games typically arises from the interplay between game mechanics, audiovisual aesthetics, and user interaction metaphors (i.e. the keys, clicks, and gestures used to interact with the game environment). Our project attempts to build upon this appeal by providing engaging games interwoven with structured pedagogy to produce new and meaningful learning experiences for new programmers.

[2] The games, source code, and all art and audio assets are freely available at https://depts.washington.edu/cmmr/GTCS/.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation grant DUE-1140410, Microsoft Research Connection, and Google under the Google CS Engagement Award. All opinions, findings, conclusions, and recommendations in this work are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] U. Wolz, T. Barnes, I. Parberry, and M. Wick, "Digital gaming as a vehicle for learning," in *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, Houston, Texas, USA, 2006, pp. 394–395.

[2] S. Leutenegger and J. Edgington, "A games first approach to teaching introductory programming," in *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, Covington, Kentucky, USA, 2007, pp. 115–118.

[3] K. Sung, M. Panitz, C. Hillyard, R. Angotti, D. Goldstein, and J. Nordlinger, "Game-Themed Programming Assignment Modules: A Pathway for Gradual Integration of Gaming Context into Existing Introductory Programming Courses," *IEEE Trans. Educ.*, vol. 54, no. 3, pp. 416 –427, Aug. 2011.

[4] K. Sung, M. Panitz, S. Wallace, R. Anderson, and J. Nordlinger, "Game-themed programming assignments: the faculty perspective," in *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, Portland, OR, USA, 2008, pp. 300–304.

[5] K. Sung, R. Rosenberg, M. Panitz, and R. Anderson, "Assessing game-themed programming assignments for CS1/2 courses," in *GDCSE '08: Proceedings of the 3rd international conference on Game development in computer science education*, Miami, Florida, 2008, pp. 51–55.

[6] R. Angotti, C. Hillyard, M. Panitz, K. Sung, and K. Marino, "Game-themed instructional modules: a video case study," in *FDG '10: Proceedings of the Fifth International Conference on the Foundations of Digital Games*, Monterey, California, 2010, pp. 9–16.

[7] C. Hillyard, R. Angotti, M. Panitz, K. Sung, J. Nordlinger, and D. Goldstein, "Game-themed programming assignments for faculty: a case study," in *SIGCSE '10: Proceedings of the 41st ACM technical symposium on Computer science education*, Milwaukee, Wisconsin, USA, 2010, pp. 270–274.

[8] K. Sung, "XNA Game-Themed Applications for Teaching Introductory Programming Courses," *Invit. Pre-Conf. Workshop Fourth Int. Conf. Found. Digit. Games Orlando Fla.*, Apr. 2009.

[9] K. Sung, "XNA Game-Themed Applications for Teaching Introductory Programming Courses," *Invit. Full Day Workshop Community Coll. Fac. Microsoft Res. Redmond Wash.*, Apr. 2009.

[10] K. Bromwich, M. Masoodian, and B. Rogers, "Crossing the Game Threshold: A System for Teaching Basic Programming Constructs," in *Proceedings of the 13th International Conference of the NZ Chapter of the ACM's Special Interest Group on Human-Computer Interaction*, New York, NY, USA, 2012, pp. 56–63.

[11] A. Luxton-Reilly and P. Denny, "A simple framework for interactive games in CS1," in *SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*, Chattanooga, TN, USA, 2009, pp. 216–220.

[12] M. C. Lewis and B. Massingill, "Graphical game development in CS2: a flexible infrastructure for a semester long project," in *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, Houston, Texas, USA, 2006, pp. 505–509.

[13] M. Külling and P. Henriksen, "Game programming in introductory courses with direct state manipulation," in *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, Caparica, Portugal, 2005, pp. 59–63.

[14] R. B.-B. Levy and M. Ben-Ari, "We work so hard and they don't use it: acceptance of software tools by teachers," *SIGCSE Bull*, vol. 39, no. 3, pp. 246–250, 2007.

[15] L. Ni, "What makes CS teachers change?: factors influencing CS teachers' adoption of curriculum innovations," in *SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*, Chattanooga, TN, USA, 2009, pp. 544–548.

[16] C. R. B. de Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson, "Sometimes You Need to See Through Walls: A Field Study of Application Programming Interfaces," in *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, New York, NY, USA, 2004, pp. 63–71.

[17] J. Stylos and B. Myers, "Mapping the space of API design decisions," in *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on*, 2007, pp. 50–60.

[18] J. Tuloch, *Practical API Design: Confessions of a Java Framework Architect*. APress, 2008.

[19] B. Ellis, J. Stylos, and B. Myers, "The Factory Pattern in API Design: A Usability Evaluation," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, 2007, pp. 302–312.

[20] B. Chau, A. Robinson, J. Pace, R. Nash, and K. Sung, "Corrupted: A Game to Teach Programming Concepts," *Computer*, vol. 47, no. 12, pp. 100–103, Dec. 2014.

[21] J. Bloch, "How to Design a Good API and Why It Matters," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, New York, NY, USA, 2006, pp. 506–507.