AC 2012-4981: EXPOSING HIGH SCHOOL STUDENTS TO CONCURRENT PROGRAMMING PRINCIPLES USING VIDEO GAME SCRIPTING ENGINES

Mr. Michael Steffen, Iowa State University

Michael Steffen is a Ph.D. candidate in computer engineering and NSF graduate research fellow. His research interests include computer architecture, graphics hardware, computer graphics and embedded systems, and specifically he focuses on improving SIMT processor thread efficiency using a mixture of custom architectures and programming models. He received a B.S, degrees in both mechanical engineering and electrical engineering from Valparaiso University in 2007.

Prof. Joseph Zambreno, Iowa State University

Joseph Zambreno has been with the Department of Electrical and Computer Engineering at Iowa State University since 2006, where he is currently an Assistant Professor. Prior to joining ISU, he was at Northwestern University in Evanston, Ill., where he graduated with his Ph.D. degree in electrical and computer engineering in 2006, his M.S. degree in electrical and computer engineering in 2002, and his B.S. degree summa cum laude in computer engineering in 2001. While at Northwestern University, Zambreno was a recipient of a National Science Foundation Graduate Research Fellowship, a Northwestern University Graduate School Fellowship, a Walter P. Murphy Fellowship, and the EECS department Best Dissertation Award for his Ph.D. dissertation titled "Compiler and Architectural Approaches to Software Protection and Security."

©American Society for Engineering Education, 2012

Exposing High School Students to Concurrent Programming Principles using Video Game Scripting Engines

Abstract

Introducing programming using an imperative language often requires a steep learning curve due to the significant emphasis and corresponding time commitment placed on a particular language's syntax and semantics. This paper presents two separate video game scripting engines focusing on nurturing computational skills that can be explored in as little as one hour. Scripting engines run code developed by students to control four concurrent players on a team; up to four teams (four different code scripts) can play in a head-to-head competition. To achieve a quick learning curve, the scripting engine only supports a limited number of instructions to define initial player qualities, movements, and game actions. Students are faced with the computational thinking challenge of mapping their game strategies into code. Successful strategies require teams to appreciate the complexities of concurrent programming to control all game players simultaneously. We have observed that students quickly learn that writing code for all team players individually does not result in a competitive match, but requires a mixture of collaboration and parallel programming to be competitive in a short amount of time. The need for more advanced control flow semantics are also motivated, since students must rewrite similar code for performing similar routines through the game simulation. The video game scripting engines have been used in two high school outreach programs and results from these events indicate that the learning objectives were met and students were engaged in the activities the entire duration by modifying their code to be more competitive. Lessons learned from the first scripting engine (`Dodgeball`) that went into creating the second engine (`Boomtown`) are also presented.

I. Introduction

With the rapid development of computing technology in consumer products, users have become familiar with computing capabilities and performance. While consumers have taken to utilizing computing technology, the majority know nothing of the development of these devices. While this disconnect is fine for consumers, it can cause problems for students who are being exposed to computer programming for the first time. Introductory programming courses spend a majority of their time covering the syntax and semantics of a specific programming language. While this knowledge is required to become proficient in computer programming, this learning method can be frustrating for introductory computer programming students, since programming semantics are usually very specific and initially difficult to understand. In addition, the outcome of a semester-long class does not always fulfill the students' expectations of creating useful or fun applications.

For use in an introduction to computer programming environment, we have created two separate scripting languages designed to teach students computational thinking and concurrent programming skills. The scripting language controls four concurrent players on a team in a video

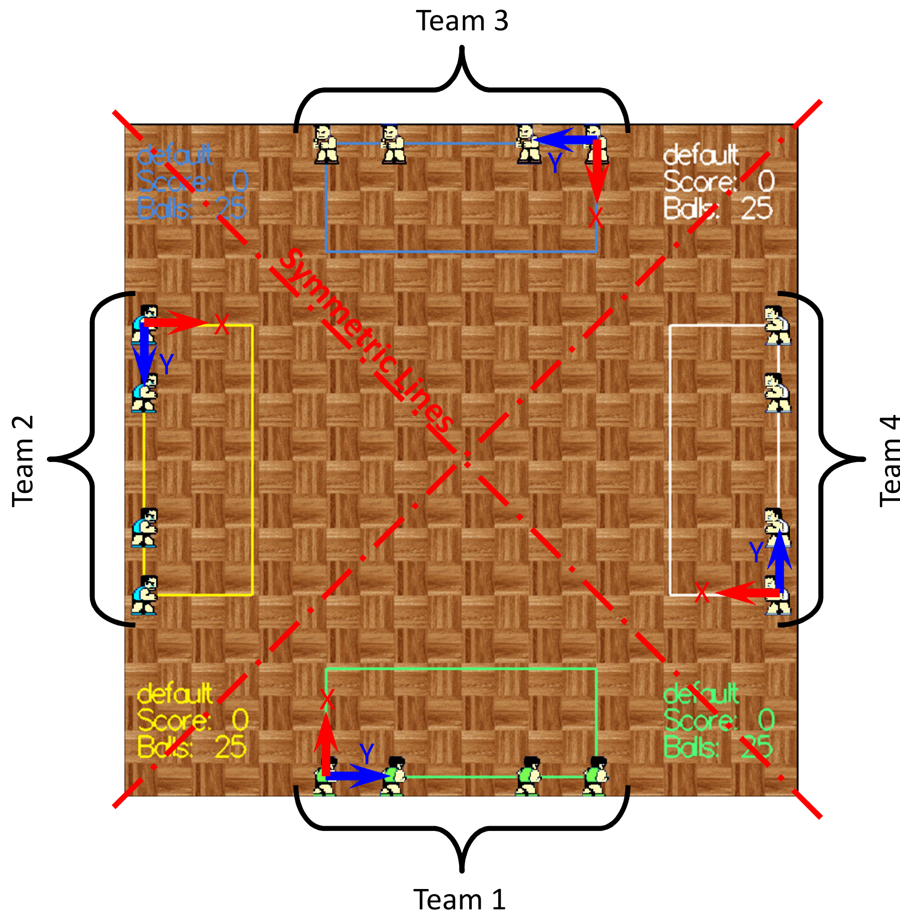

Figure 1: Our first video game playing field (`Dodgeball`), where students control four concurrent players on a team using a custom scripting language.

game where teams can play in a head-to-head competition. Both of the scripting languages are specifically for their video game. A limited number of instructions are implemented for controlling initial player status, position and game actions. By using a limited number of instructions, the challenge for students is coding up a game strategy and working efficiently in teams, rather than learning syntax. When student code is executed, a simulation of the video game is also displayed on the monitor for students to watch, keeping them engaged and connected to their familiar experiences with interactive applications. Ultimately, the intended goal for these scripting engines are to get students with no coding experience excited about computer programming by introducing computational thinking problems, rather than bogging them down with coding semantics.

Both video game scripting engines have been used during two separate high school outreach programs. Students were able to pick up on the coding method and implement competitive strategies in as little as one hour. In addition, students as young as sixth grade have been able to implement their own strategies using our scripting language.

The remainder of this paper is organized as follows: Section II presents a brief overview of related environments used to teach computer programming. Section III describes both of our scripting engines and provides details about the video games implemented. Sample students implementations are shown in Section IV, and Section V presents lessons learned from our first game engine. Section VI concludes the paper with a discussion of planned future work.

II. Related Applications

Mindstorms[4] are a LEGO-based construction kit that allows for students to build and program simple robots. LEGO Mindstorms is targeted for kids aged ten years and older, and supports multiple programming languages from conventional text-based to graphical flow charts. A wide range of motors and sensors are available, allowing for a large variety of projects and coding challenges.

Alice 3D[1] is a programming environment that allows students to create 3D applications. Students can use Alice 3D to tell stories, create games and produce movies. Programming in Alice 3D uses a graphical interface, where students drag-and-drop instructions, objects and actions into their application scene. By using a graphical interface that allows drag-and-drop of instructions, students are not weighed down by language syntax. MIT Scratch[2] is another environment for creating videos and games using a drag-and-drop programming method. Processing[3] is another programming language for teaching programming concepts using a visual style. Students develop applications used to generate artistic scenes and simple games.

These examples provide a wide variety of functionality for people interested in learning how to program, but can require some time (on the order of hours for Alice and Scratch, and days for Mindstorms and Processing) before introductory students can become proficient enough to create their own interesting designs.

III. Video Game Scripting Engine Framework

Both video game scripting languages interface with video game environments consisting of 2D playing fields, four team members and a total of four teams. Figure 1 shows our first video game (`Dodgeball`) in action. The major differences between the two engines are the video games being played, requiring different game action instructions and display engines.

Students first develop their application using a text editor to control one team. Each source code file controls instructions for four players on the same team. The code to control a single player is broken down into two parts: an initialization instruction and instructions executed during the actual game play. The first instruction for every team member must be an initialization function (similar to allocating a variable). The initialization instruction takes a single argument that defines the type of player. Multiple pre-defined player types can be chosen, each with different strengths and weaknesses. After the initialization is processed, the game starts, and player instructions are executed sequentially. Once the current instruction for a player is completed, the next instruction is executed. To simplify the programming model, no control flow instructions are supported (e.g.

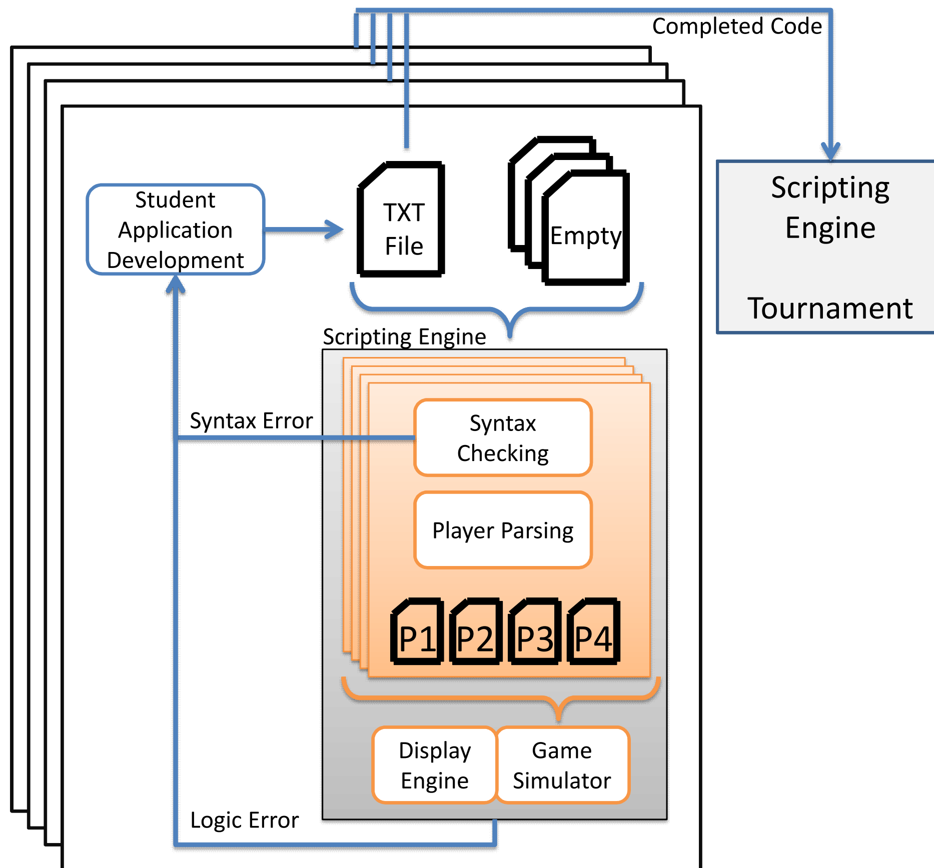

Figure 2: General development flow for students programming our video game scripting engines.

if/else, loop, goto), and there are no instructions that allow teams to know the status of any other player.

The scripting engines allows up to four team scripts to be combined in a head-to-head competition, requiring different starting locations on the playing field. To allow for the scripting strategies to be position-independent, the game board is symmetric across the two forty-five degree symmetric lines. In addition, all commands are relevant to the starting coordinates. Starting coordinates for each team's starting location are positioned such that the same strategy is executed.

The script is then simulated by running the executable for the scripting engine, which is composed of three parts: compiler, game simulation engine and graphics rendering engine. Figure 2 illustrates the student script development process, from initial development to execution of finished scripts in a head-to-head competition. The compiler reads up to 4 script files (one for each of the teams) and checks for syntax errors. Because only a limited number of instructions are supported, compiler error messages attempt to be as specific as possible, and provide example valid syntax for many common problems. Once the syntax check is finished, each of the team codes are divided into individual player codes that are fed to the concurrent game engine processors.

Table 1: Different player types for the `Dodgeball` scripting engine

Player Types				
Name	Speed	Time between throws	Catch Angle	Max Catch
Attacker	Fast	Cheetah	20	2
Defender	Fast	Moderate	60	4
Enforcer	Moderate	Moderate	20	1
Goalie	Moderate	Moderate	90	3
Sprinter	Cheetah	Fast	20	2
Pest	Slow	Slow	140	8
Captain	Fast	Fast	20	3

The game simulation engine keeps track of all player states and applies the rules of the game. The simulation engine also keeps track of the score in real-time and disables individual player processors when game actions force them out of the game. While the simulation is running, the display engine renders the playing field with all players and the current score. The visual display allows students to debug their code by following a player in the display and referring to their source code. The game simulation can be played at a normal speed, paused, or have the time incremented by a small amount for every key press.

III.A. Dodgeball Game Engine

In `Dodgeball`, the first video game scripting engine, students control the 2D position of the player in a gym, including their orientation (direction facing), and state (either throwing a ball or catching). Teams receive a point for every opposing team member they hit with one of their balls and also for every ball they catch thrown by an opposing team. If a player is hit or their ball was caught, they are out of the game for the current round (no longer able to execute instructions). If a ball thrown by a player's team hits another member of the same team, the team member hit is out and that team loses a point. Each team starts with twenty-five balls that are shared among all team members. To increase the likelihood of balls hitting players in the open field, balls are allowed to bounce off walls (including the bounding walls), where the number of bounces is set by a parameter of the different dodgeball players students can select from.

When implementing a strategy, students must decide on the type of dodgeball players to use and how they should play the game. Table 1 shows the different dodgeball player types and Table 2 shows the instructions they can use. A sample program is shown in Algorithm 1. In this program, a *Captain* dodgeball player is defined. When the game starts the player moves to the 10,10 cell in the playing field. The origin of the coordinate system is determined by the starting location (see Figure 1). Then the player turns 45 degrees to the left and throws a ball. Degree angles are absolute angles such that 180 degrees is parallel to the positive X axis of the coordinate system. After that move, the player turns 90 degrees to its right and goes into catch mode, now being able to catch a ball that would hit this player if it comes in at the right angle.

Table 2: The instructions for the Dodgeball scripting engine

Dodgeball Instructions		
Syntax: P[Player Number].[Instruction] {Arguments}		
Instruction	Arguments	Description
Create	Player Type	Creates a player
Move	X,Y Position	Moves player around the game field
Turn	Angle (Deg)	Rotates the player to face a new direction
Throw	None	Throw a ball in the direction the player is facing
Defend	Time (Sec)	Player waits to catch a ball

Algorithm 1 Example script for Dodgeball

```
1: P1.Create Captain
2: P1.Move 10 10
3: P1.Turn 135
4: P1.Throw
5: P1.Turn 225
6: P1.Defend 500
```

III.B. Boomtown Game Engine

The second scripting engine is named Boomtown (Figure 3). Students move players around on a 2D playing field. Players can drop bombs onto the playing field that explode after a set amount of time. Bombs explode a set amount of spaces in the vertical and horizontal direction. Any player that is in the range of the bomb is out (cannot execute code). Similar to the Dodgeball example, a point is won for every opponent team member hit by one of the players team's bombs, and a point is lost for every team member hit by one of the player team's bombs.

Students have multiple options for implementing a strategy from the different types of player characteristics and instructions they can use (see Tables 3 and 4). Instead of using absolute values for movement and orientation, relative values are used. Movement values are limited to whole numbers and orientation only allows left, right and reverse direction changes. To allow for independent starting positions, the game field is also required to be symmetric across the two 45

Table 3: Different player types for the Boomtown scripting engine

Player Types				
Name	Speed	Bomb Radius	Bombs	Safe From Team Bomb
Captain	Default	Default	Default	False
Punter	Slow	Default	Less	False
Stretcher	Slow	Big	Default	False
Safety	Default	Small	Less	True
Sprinter	Fastest	Default	Less	False
Hoarder	Slowest	Default	Most	False
Hulk	Slow	Default	More	False

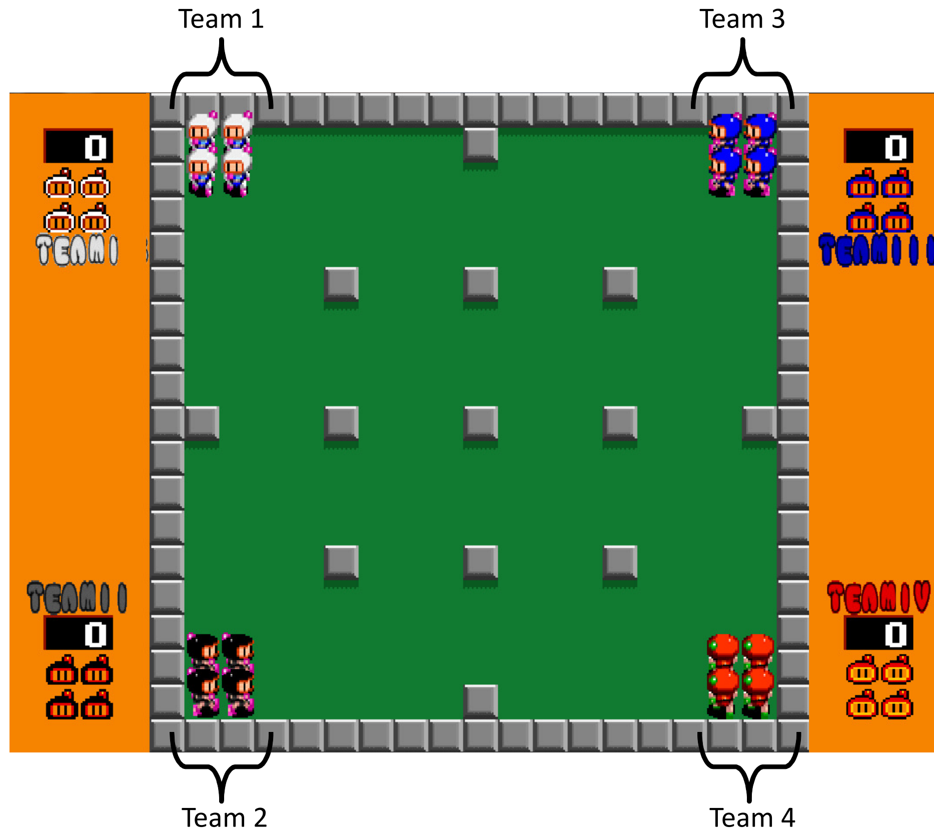

Figure 3: The second video game playing field where student control four concurrent players on a team using a custom scripting language.

degree symmetric lines and starting location and orientation.

Algorithm 2 shows a sample program for `Boomtown`. In this example, a stretcher character is defined. The player moves forward three spaces, turns left, moves forward two spaces, drops a bomb, turns right, and then moves forward six spaces. The bomb exploded while the player is in the process of the last move instruction, but the player has moved enough spaces away from the bomb to avoid the explosion.

III.C. Concurrent Programming

One learning objective of the scripting engines is for students to learn the challenges of implementing their ideas in code, rather than struggling with complex syntax. The computational thinking skills used for the scripting languages focus on students developing a strategy and then transferring that into code. To keep students challenged with implementing different strategies, the playing field can be customized. Customizing the playing field for both scripting engines involves changing wall locations, creating alternate playing fields and adding or removing obstacles. Concurrent programming is introduced by having four independent players per team. While the code for all players is in a single file, each player code is executed in parallel. This

Table 4: The instructions for the `Boomtown` scripting engine

`Boomtown` Instructions		
Syntax: P[Player Number].[Instruction] {Arguments}		
Instruction	Arguments	Description
Create	Player Type	Creates a player
Left	None	Turn to face left of current orientation
Right	None	Turn to face right of current orientation
Reverse	None	Turn to face the opposite direction
Move	Number of Spaces	Move the player forward
Drop	None	Drop a bomb at current position
Wait	Time (Sec)	Player does not move for time specified

Algorithm 2 Example script for `Boomtown`

```
1: # Define player 1
2: P1.Create Stretcher
3: P1.Move 3
4: P1.Left
5: P1.Move 2
6: # Drop a bomb
7: P1.Drop
8: P1.Right
9: P1.Move 6
```

allows up to four instructions to be executed for a team at one time. The location inside the code for each team player is not relative, allowing code for specific players to be grouped together in the source code (allowing for easier copy and paste from programmers working in parallel on different team players) or interleaved for better understanding of how two players may move in parallel. To help stress the importance of communication with concurrent programming, game actions used for increasing a team's score when encountered by other teams will decrease the team score if the action encounters someone from your own team. This requires students to not only understand the complexity of working efficiently (coding team players in parallel), but also having to stay in communication so players on the same team do not disrupt one another.

IV. Example Student Results

Both scripting engines have been used in separate high school outreach events.[5] For both events, approximately forty students participated in ten groups of four students each. The teams were provided with a sample program and handout describing the scripting language and how to run the application, along with the game engine configuration file. Students had one hour to become familiar with the scripting engine language and implement a strategy. Then a round-robin tournament was held. The round-robin tournament gave students the opportunity to study other team's strategies, evaluate their standing against other teams, and make changes to their code before competing in an elimination tournament. The results of the round-robin tournament were then used to create a tournament bracket. Each round of the tournament bracket used a different

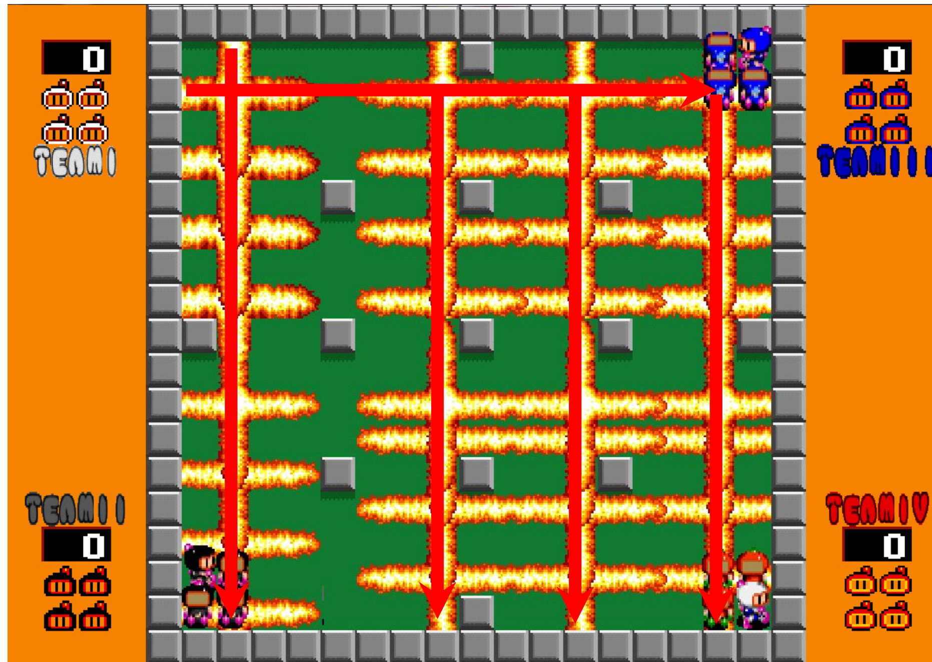

Figure 4: A strategy that used the four concurrent players to cover as much of the playing field as possible.

scripting engine configuration. Students had an additional thirty minutes to implement a strategy if they passed to the next tournament round.

In our first example (from the `Boomtown` competition), a student team used the four concurrent players to place bombs in locations that completely covered the playing field (see Figure 4). This strategy divided the playing field into four sections, moving one of the players to each section. The players then moved from top to bottom, evenly distributing bombs along the way. This team selected the optimal players for speed, number of bombs and the bomb explosion width. In this strategy, the team had a large coverage of the field, but any players missed in this single attack could not be attacked again.

In the second example, a student team used three of their players in an attempt to create a wall of explosions, such that any player entering that area would have a high chance of being hit. The team positioned their players (see Figure 5) for the wall, dropped a bomb, and then moved back to a safe spot. This strategy was continually repeated until they ran out of bombs. The fourth player was a safety player that continually dropped bombs in the starting position. After the round-robin tournament, students noticed that a majority of the teams sent players to the starting position of other teams to score points off players that did not move far. Teams later countered this strategy by keeping a player in the starting location to continually drop defensive bombs.

In both strategies presented, students realized that they were re-using part of their code over and over again. While they were able to use copy-and-paste methods, they realized that looping constructs (e.g. for/while) would be beneficial and time saving.

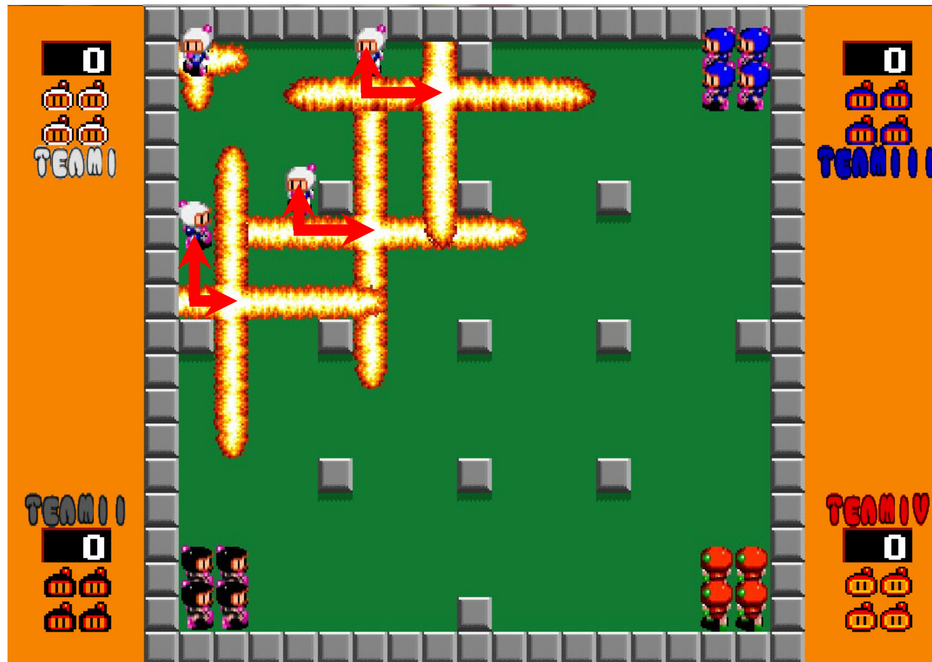

Figure 5: A defensive strategy that used four concurrent players to create a wall to hit any approaching opponent players.

In addition to high school students, multiple undergraduate classes have also used and help develop the scripting engines. A senior level computer architecture course was used to stress test the scripting engines during development in a way that taught students about designing tests for stressing their own designs and also creating tests that simplify debugging logic errors. A freshman-level introduction to programming course also used the source code for the scripting engine to connect what students have learned to a more complex distributed application, allowing them to add their own features to the scripting engines.

V. Lessons Learned

The second scripting engine (`Boomtown`) was created after the first outreach program using `Dodgeball`. During this event, we observed some areas that could be improved to increase student learning. First, the flexibility in the syntax checker was improved to generate more helpful syntax error messages. These improvements gave students an easier time debugging their own syntax errors, allowing them to pay less attention to the syntax rules.

The second observation was to implement a second video game that allowed for more complex strategies, and therefore more intentional coding. Observations made from `Dodgeball` showed that successful strategies were ones where students just threw as many balls in random directions as possible. The only challenging part was for students to make sure that the balls thrown did not hit their own team members. In the development of `Boomtown`, a key design element was to improve on strategy planning. Consequently in `Boomtown`, game actions (such as bombs exploding) take place in only small areas of the playing field, unlike throwing balls that can go all

over the playing field. This required students to move their players around the playing field and be selective on their game actions. As a result, we noticed more elaborate game strategies implemented in `Boomtown` than in `Dodgeball`.

After the second outreach event using `Boomtown`, additional opportunities for improvements have been discovered. Current implementations require students to develop their code in a separate file and then open it in the scripting engine. Developing an Integrated Development Environment (IDE) will streamline debugging and running programs. Debugging logic errors are also difficult since there is limited correlation between the source code and game simulation. Further debugging options that address these problems will be implemented in future scripting engines.

VI. Conclusion

This paper presents a framework for video game scripting engines designed as an introduction to computer programming. Students, with little to no experience in programming, write code using a limited set of instructions to control game characters in a head-to-head video game competition. Since only a few instructions are available, students quickly learn the language syntax and semantics and are able to focus their time on computational thinking skills. In our example scripting languages, students are faced with the challenge of implementing a game strategy in code. For students to have a competitive strategy, they must understand the challenges of team programming and concurrent programming. The scripting languages have been used in high school outreach programs where we have observed students successfully understanding the challenges of working on concurrent programming to implement competitive strategies. Students' written comments from the outreach event regarding the scripting engines were positive and enthusiastic. Several students even requested copies of the scripting engine. School children as young as sixth grade have also been able to develop strategies using the scripting engines. Both the `Dodgeball` and `Boomtown` video game scripting engines are available for download at our research group's website.[6]

References

[1] Matthew Conway, Randy Pausch, Rich Gossweiler and Tommy Burnette. "Alice: A Rapid Prototyping System for Building Virtual Environments". *IEEE Computer Graphics and Applications*, vol. 15, pages 8-11, 1994.

[2] John Maloney, Leo Burd, Yasmin Kafai, Natalie Rusk, Brian Silverman and Mitche Resnick. "Scratch: A Sneak Preview". *Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing*. 2004.

[3] Casey Reas, Ben Fry. *Getting Started with Processing*. Make, 2010.

[4] Dave Baum. *Definitive Guide to LEGO MINDSTORMS*. Apress, 2002.

[5] Julie Rursch, Andy Luse, Doug Jacobson. “IT-Adventures – A Program to Spark IT Interest in High School Students using Inquiry-Based Learning with Robotics, Game Design, and Cyber Defense”. *IEEE Transactions on Education*, Vol. 53, Issue 1, pages 71-79, 2009.

[6] Reconfigurable Computing Laboratory, Iowa State University, http://rcl.ece.iastate.edu/