A Zero-sum Game in Lua

Zackery L. Arnold
Department of Electrical and Computer Engineering
University of Dayton
Dayton, Ohio 45469–0232 USA
arnoldz3@udayton.edu

ABSTRACT

The system presented is a competitive game written in the Lua programming language using the LOVE2D game engine. The objective of the game is to best an opponent in a competition based loosely upon the ruleset of the traditional game *rock-paper-scissors*. Each player controls a character that is capable of changing between a fire, water, and grass form in the style of classic games, where each form is represented by three colors. In the rock-paper-scissors tradition, fire beats grass, water beats fire, and grass beats water. When the players make contact, the player with the winning color earns a point and the other player is reset to a random position on the board. In addition, environmental hazards of a given color are present on the field, adding a need for positional awareness. The first player to earn a majority of points wins the game.

KEYWORDS

Callbacks, delta-time, LOVE2D engine, Lua programming language.

Zackery L. Arnold. 2017. A Zero-sum Game in Lua. *CPS 499-03: Emerging Languages, University of Dayton Ohio 45469–0232 USA, Spring 2017,* 3 pages.

1 INTRODUCTION

We developed a competitive game that was developed using the Lua programming language (https://www.lua.org/) and LOVE2D (https://love2d.org/)—an open-source 2D game engine. Game development projects often provide exciting challenges for new programmers and serve as excellent entrance points into the study of software engineering [1]. Individuals learning about game programming likely encounter many diverse fields of computer science, including artificial intelligence, networking, and human-computer interaction. Computer games are also, if well-designed, fun and visually stimulating projects that help to maintain focus and attention throughout development.

Lua and the LOVE2D engine were used and serve as quick and effective tools in rapid game design prototyping. The LOVE2D engine, in particular, features many libraries, including those for graphics, sound, and user input, that allow the programmer to quickly move from a conceptual design to a prototypical implementation.

CPS 499-03: Emerging Languages, University of Dayton Ohio 45469–0232 USA

2 IMPLEMENTATION

All games created with LOVE2D operate through the use of user-defined versions of callback functions. The `love.load` callback function, for example, is invoked when the application launches. This system uses this function to initialize the state of the game by assigning the player's gamepad to one of the two characters, initializing the game window with a widescreen resolution, and starting the background music track on a constant loop with the help of the `love.audio` library. The debug, scoreboard, game state, player, and map tables are also initialized through use of an `initGame` function that can be called again during gameplay.

The `love.update` function is invoked every time a new frame is generated for the game. The implementation for this system starts by checking if a game is paused or completed, halting the rest of the function's calculations until a game is resumed. If the game is still running then the function checks if a win condition has been met and, if so, sets the appropriate state flags. If the game is still in play then the function is responsible for handling player movement input and executing collision detection to determine if either player character is in an damaging position.

Finally, the `love.draw` function is responsible for drawing graphics to the screen for each new frame. This system constantly draws the map, player characters, and scoreboard for each frame generated.

2.1 Collision Detection

Player positions are maintained using the `player.x` and `player.y` values, each of which are assigned to specify an `(x, y)` location in the coordinate plane relative to the origin in the top left corner of the display. For every new frame, the game's update loop must check if either player is intersecting a hazard or the other player. The obvious way to do this, given that both the player and the hazards are defined as simple rectangles, is to check if any of the four corners of the player intersect one of these illegal zones. Problems arise, however, when random location spawning is added. The players could spawn at a location on the map in which a hazard was passing through the center of a player between all four of the player's corners.

To resolve this issue, a function called by the game update loop checks for one of four conditions that implies that the player is **not** colliding with another object in the environment. For example, due to the simple geometry of the player characters, if the rightmost edge of the first player is left of the leftmost edge of the second player, then the two players could not possibly be intersecting. This simplification is only possible due to the simple geometry of the player characters. More challenging representations of player characters, such as three dimensional models, necessitate highly complex collision detection functions.

Table 1: Cell values and their gameplay representation.

Cell Value	Graphical Representation
`0`	Empty or blank space.
`1`	White impenetrable wall.
`"w"`	Water-type environmental hazard.
`"f"`	Fire-type environmental hazard.
`"g"`	Grass-type environmental hazard.

2.2 User Input and Player Movement

LÖVE2D features an easy-to-use interface for handling user input. For example, whenever a key is pressed by a player, an event occurs that is immediately handled by the program's implementation of the `keypressed` callback function. The key can then be identified and used to trigger other specific events in the program. In this system, the `F2` key is used to start a new game and the `F10` key is used to display a debug page that prints various values of interest during runtime. Additionally, the `up`, `down`, `left`, and `right` keys are used for movement of the first player while the `w`, `s`, `a`, and `d` keys are used for the movement of the second player. Finally, the right `ctrl` key and `spacebar` allow each player to switch between their three forms represented by their respective colors. If a standard video game controller is connected before runtime, then each of the functions are also mapped to logical buttons on the controller.

The feeling of movement in a computer game is one of the most subtle, but important, aspects of game design. A balance must typically be found between movement controls that are far too sluggish and those that are far too quick as either scenario can alienate a new player. In this game, player movement is accompanied by a period of acceleration before the maximum speed of the player is achieved. Deliberate acceleration is also useful as a means of influencing the tension of a competitive game such as this, where unnecessary stops by a player can cause them to be caught when they have to regain their momentum. Initial iterations of this simple movement system only involved a linear acceleration value applied to the `player.speed` value. However, different computers could possibly run the game at different frame rates for various technical reasons. This means that the different computers were updating the game at different rates, resulting in different control experiences. To resolve this issue the game logic must rely upon the passage of real-world time instead of what is known as *frametime*. This is achieved by including a `dt` or *delta-time* value in calculations. This term, common within the game-programming industry, refers to the time that elapses between two adjacent frames of the game. A delta-time value is automatically generated by the `love.update` callback function.

2.3 Map and Hazard Generation

Hazards are generated throughout the game map by the system's map generation table. When a new game begins the map table's `generateMap` function is called with three arguments—the width of the map, the height of the map, and the square size of each cell within the maps' bounds. The function then constructs a blank table to the given dimensions with each cell initialized to zero. The outer perimeter is constructed first by setting the cells along the border to a value of 1 which represents an impenetrable wall. Then a random cell location is specified for the starting point of one of the three environmental hazards. This cell is assigned an elemental value that corresponds to the current hazard. These values are listed in Table 1. Random directions for the expansion of the hazard are chosen through use of a loop that iterates for each additional cell of the hazard. If the next cell in the chosen direction from the current cell is empty then the cell is assigned the hazard's value. This process then repeats for the other two remaining hazards. Once all hazards have been created within the table the map is returned to the system and instantiated as the playfield. Player collision in this instance is handled through corner checking as the map table can be easily indexed by each of the four corners of a given player's location.

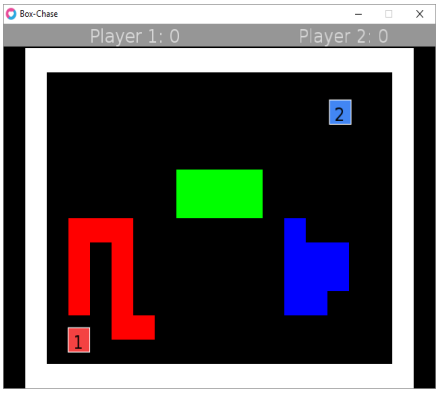

Figure 1: An example map layout of the game.

2.4 Graphics Drawing

Fig. 1 illustrates an example screenshot of the game featuring both of the player characters as well as the three environmental hazards within the map. The `love.draw` callback function is responsible for drawing graphics onto the game window. The order in which the graphical elements are drawn determines the way in which they are viewed upon the screen. The map, for example, is rendered before the player characters so that both characters remain visible when they are safely passing over an environmental hazard. Similarly, a number is drawn after each player character to provide a way for the player to visually distinguish of which character they are in control. Balancing the order with which the items on the screen are drawn is an important part of the LÖVE2D development process.

The SUIT library used with this program manages its own drawing process with the `suit.draw` method. However, the objects created with SUIT must be instantiated to specific locations within the game region. SUIT's drawing process is therefore non-conforming with the regular workflow of graphics production in LÖVE2D.

3 CONCLUSION

We presented a zero-sum game prototype developed in Lua with the LÖVE2D game engine. Each player is able to independently move across the playing field while the game logic constantly checks for any collisions. Full keyboard and gamepad support is implemented for the control scheme providing multiple and convenient ways for

the players to experience the game. Improvements to this prototype including code modularization so to better exploit the strengths of Lua tables. Additional features, such as advanced movement (e.g., jumping), that provide players with more interesting gameplay opportunities can also be incorporated.

REFERENCES

[1] K. Claypool and M. Claypool. 2005. Teaching software engineering through game design. In *Proceedings of the Tenth Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM Press, New York, NY, 123–127. Also appears in *ACM SIGCSE Bulletin*, 37(3), 2005.