Alex Owens

Dr. Jeff Ward

HNR 491

14 April 2019

Puzzler: A Programming Language For 2D Puzzle Games

Abstract

Most popular computer programming languages are designed for implementing a wide array of applications, and are consequently very complex. Conversely, Domain Specific Languages (DSLs) are much smaller languages designed to make creating one type of application easy. Puzzler is a DSL that streamlines, for both programmers and non-programmers, the creation of 2D grid-based puzzle games. Non-programmers can use it to make meaningful changes to their games much more quickly than in a complex, general purpose language. Puzzler is implemented in the Racket programming language, a current Lisp dialect with a great deal of direct support for programming language creation.

I. Introduction

The field of computer game development is massive. It is the marrying of many seemingly unrelated disciplines such as computer science, art, music, and literature. As

a computer scientist my interests lie on the technical side, the computer programming, of creating these works of interactive art. Mainstream game programming has long been dominated by one programming language: C++, which is chosen for its ability to be fast and close to the hardware on which the game is running. As a novice programmer or game creator, the mountain that is learning C++ is nearly impossible to overcome. Even very experienced C++ programmers often want to write game logic at higher levels of abstraction than C++ can provide. Because of this there has been a lot of development effort put into creating higher level tools, which open the field of game development to those who see learning C++ as too big of a time investment. Puzzler, a domain-specific programming language for creating 2D puzzle games that I designed, is one such tool. This essay discusses the impetus for creating Puzzler, as well as many of the design problems I had to solve while working on Puzzler as it is today.

II. Motivations for Creating Puzzler

When video games first started getting popular with things *Galaga* in 1981, games were programmed using very low-level assembly language (Midway Games 5-3, 5-5). This required game programmers to move values in and out of computer registers and know how to interact intelligently with precise memory locations on their computer. Over the years higher level tools started gaining popularity for game programming. For example, *Half Life* in 1998 was written in C++, and C++ has stayed a very dominant force in the game programming industry since about that time (Valve Software).

The use of C++ in modern game programming is a double-edged sword. As an Object-Oriented language, C++ provides very high level abstractions for programmers who do not need to see the low-level computer architecture. But when one does need to interact with things like memory layout and CPU cache locality, C++ provides tools for that as well. All of this freedom comes with a price, however, and it is one that new programmers and experienced programmers both must pay. C++ is a very complex language that can take years to learn well. As someone getting into the field of game programming as a career, this is not a huge ask, but this is outrageous for those who just want to make video games in their spare time for fun. My language, Puzzler, is one solution to this problem, and it aims to help those who have little or no programming experience create games quickly and easily.

III. The Puzzler Programming Language

The primary purpose of Puzzler is to make creating grid-based 2D puzzle games easy for those who have no programming experience. When designing the language I tried to keep this goal in mind as much as possible. Figure A gives the source code for a game written in Puzzler. This is a game about pushing blocks and trying to get to an image of a chick on the map (see Figure B). The player can move using the arrow keys and when they move into a space with a block, the block is pushed in the direction the player is moving.

```
1  #lang puzzler
2
3  START_MAP
4  BBBBBBBB
5  BP#####B
6  B######B
7  B####BBB
8  B###B##B
9  B###B#CB
10 B###B##B
11 BBBBBBBB
12 END_MAP
13
14 -- Takes care of where to draw things
15 draw:
16   "P" -> "player.png"
17   "C" -> "chick.jpg"
18   "B" -> "rect"
19
20 -- Takes care of how to respond to key inputs
21 action:
22   "P": "up" -> (0, 1)
23   "P": "down" -> (0, -1)
24   "P": "left" -> (-1, 0)
25   "P": "right" -> (1, 0)
26
27 -- Takes care of when player is trying to move into block position, will push
28 interactions:
29   "P" push "B"
30   "B" stop "B"
31   "P" grab "C"
32
33 -- Takes care of win conditions - maybe be more explicit with positions
34 win:
35   "C" count_items 0
```

Figure A: Source code for the Puzzler game "Free the Chick"

There are other rules in the game, for example the player cannot move out of the map or push sequences of multiple blocks. This is the entire source code for this small puzzle game, which as you can see is 35 lines including spaces and comments. The exact same game written in C++ equated to a little over 350 lines of code.

The first line of this program tells the Racket compiler that we are going to use the Puzzler language (which itself is implemented in Racket). Lines 3-12 are what I call

the "map section": in it all of the maps in the game are defined by lines filled with capital letters or symbols. This game only has one map, but to create games with more maps, you simply have to add a blank line and then start your next map. On the map we have an assortment of various letters like "B", "P", and "C". These letters represent entities in the game such as the blocks, player, and chick. To change the position of any given entity, all that the programmer has to do is move a letter on the map. This acts as a nice visual representation of the level being designed. It is also important to note that "#" is a reserved symbol in Puzzler maps, and it denotes an empty space. Line 14 is the first example of a comment in Puzzler. A comment is simply a line that does not get turned into executable code and is ignored by the compiler, so they are used for humans to increase readability. Lines 15-18 defines the "draw section", which starts with the phrase "draw:", and is followed by "draw rules". A "draw rule" tells the program what to draw when it sees a specific letter on the map. For example, the first draw rule maps the letter "P" to "player.png", which is the name of an image file on the computer that is running the program. The rest of the draw rules behave similarly. Lines 21-25 make up the "action section" for the game, which defines various "action rules". The "action rules" in a Puzzler game define the player input from the keyboard, which is the only kind of input that Puzzler has in its current state. The first rule, on line 22, tells Puzzler that whenever the "up" arrow is pressed on the keyboard, any "P" entity (the player) will move zero in the x-direction and one in the y-direction, which means straight up by one space. The "action rules", such as "P": "up" -> (0,1), define rules that take entities that exist on the map (in this case, the player 'P') and any keyboard key (in this case the up

arrow), and move the entity the specified number of tiles in the x direction and the y direction. Lines 29-31 are the "interactions" in the game. "Interactions" in Puzzler are a way to represent game mechanics. For example, the first interaction rule, "P" push "B", says that when the player (P) moves into a space with a block (B), that block will be "pushed" by the player. There are other interactions such as "stop", which blocks movement into a space, and "grab", which says that when the first entity moves onto the second entity's square, it takes over that square. In the case of our "Free the Chick" game, this is how the player "frees" the chick. The player has a grab interaction with the chick that removes the chick from the map after the player has touched it. The final lines of this game, 34 and 35, define the "win rules" of the game. In this case there is only one win rule, which checks to see if there are any more chicks on the map. The rule "count_items" takes a number and if there are exactly that number of the entity on the map, then the game is considered won.

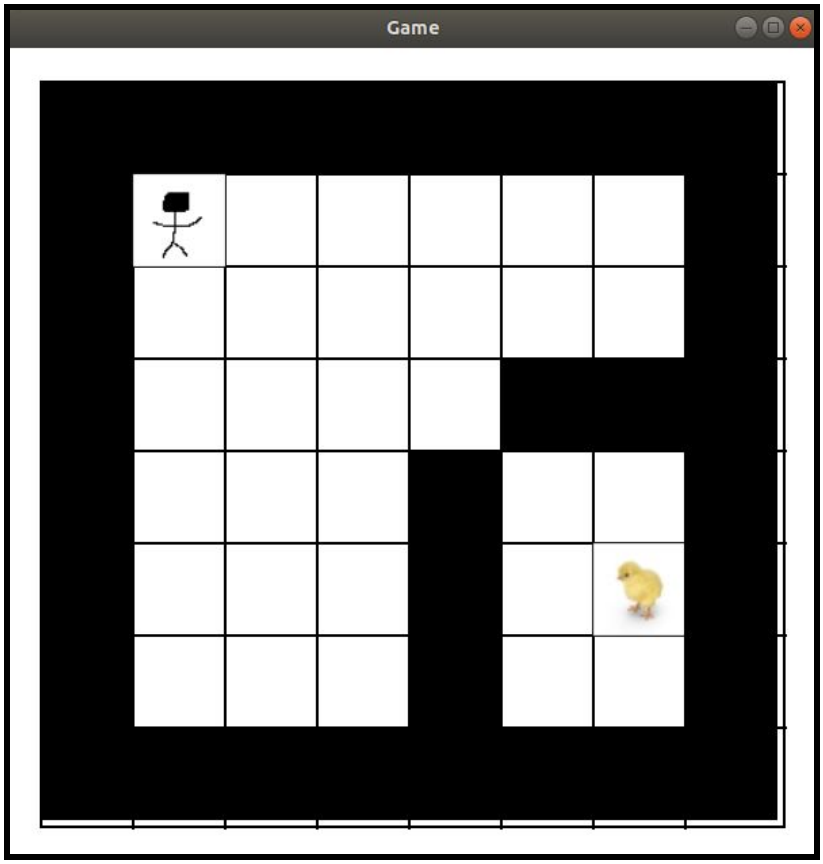

Figure B: The running "Free the Chick" game on Ubuntu 18.04

There are other constructs in the language such as the "goal map" and "events".

IV. The Design of Puzzler

When I first set out to design a programming language for games, I made one huge mistake: trying to be too general. The computer programming field is filled with general purpose languages which can be used to nearly anything you want. My goal was to make it easy for people to make games, and so in trying to design a language for that, I realized that I needed to be more specific. This is when I decided to focus on 2D puzzle games. There were a number of factors in this decision: given a finite set of rules and

mechanics, one can often express a good variety of new puzzle games, and so the puzzle genre seemed to be an easy target for generality. I wanted the language to be simple for new programmers and non-programmers, so 2D games were an obvious choice as 3D games generally require a lot of technical math, are more complex, and require labor-intensive creation of art assets.

The map section in Puzzler came about because I wanted designing levels to be simple and fun. The graphical representation of a map as a grid of letter was directly inspired by another game-making tool: Stephen Lavelle's *PuzzleScript*, which provides a similar way of representing game maps (Lavelle, "PuzzleScript"). I wanted newcomers to be able to make easy, meaningful changes to the game maps and this seemed like the best way of representing them visually.

The real power of Puzzler comes from "interactions" and "win rules", and these constructs are how the real semantics of a game are expressed. The interaction section of a Puzzler program describes game mechanics. For example, the line "'P' push 'B'" means the player can push blocks around in the game world. I wanted it to be very easy to add these rules and change them, so all a user has to do is change a verb such as "push" to a different verb in order to change the mechanics of the game. I wanted win rules to be equally as concise and expressive, so I added constructs like the "goal map", in which players can define a map which the game checks against the current map to see if the game is won. To change the win state, one simply has to change the "goal map". There are also lose rules in Puzzler, which define the loss state of a game, and all of these

lose rules are interchangeable with the win rules. To find an exhaustive list of all interactions, win rules, and lose rules, see the appendix.

V. *Additional Game Examples*

Puzzler is a tool that can be used to express many different kinds of games. The next example, shown in Figure C, is an implementation of Sokoban in Puzzler.

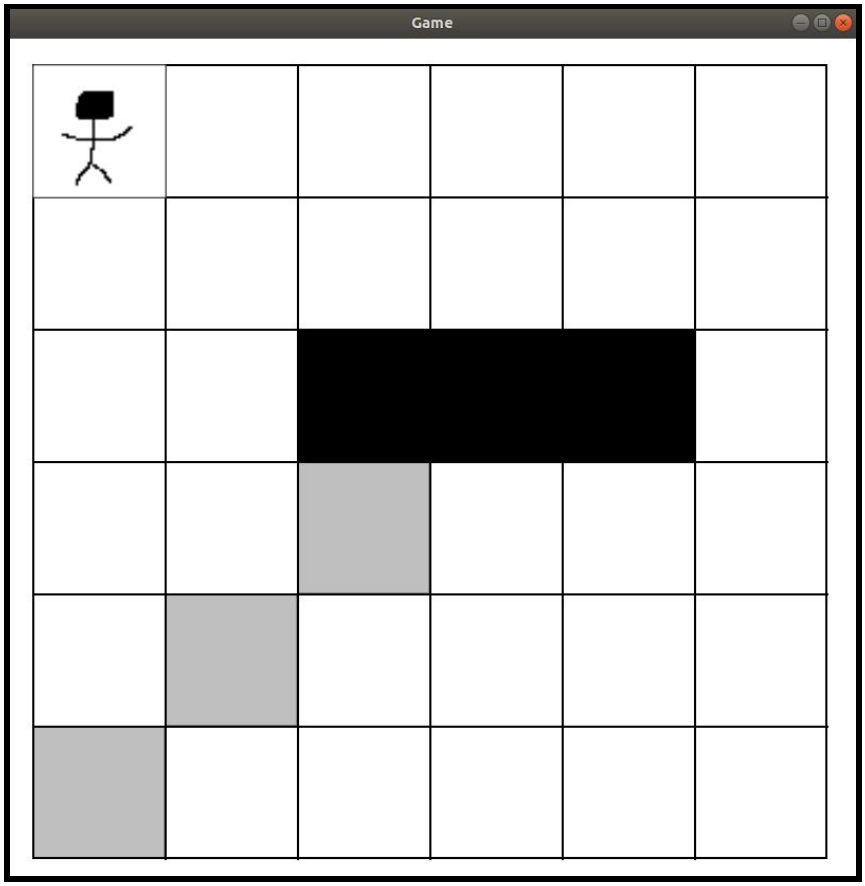

Figure C: Sokoban implemented in Puzzler

Sokoban is a type of puzzle game where the player has to arrange blocks in a certain order. For example, in this game the player has to push the three black blocks into the

grid spaces that are grey. Sokoban is a game that has been implemented in numerous programming languages and Puzzler's implementation (Figure D), is very concise.

```
#lang puzzler

START_MAP
P#####
######
##BBB#
######
######
######

####P#
##B###
######
######
######
######
END_MAP

START_GOAL_MAP
######
######
######
##B###
#B####
B#####

######
######
######
######
######
###B##
END_GOAL_MAP

draw:
  "P" -> "player.png"
  "B" -> "rect"

action:
  "P": "up" -> (0, 1)
  "P": "down" -> (0, -1)
  "P": "left" -> (-1, 0)
  "P": "right" -> (1, 0)

interactions:
  "P" push "B"
  "B" stop "B"
```

Figure D: The Puzzler code for Sokoban with two levels

Another game I implemented during the development of Puzzler is one about collecting items from specific spots on the grid. The puzzle in this game is that every time you move, a block fills in the space you were occupying. Since the rules for the game state that a player cannot move blocks around, the player must figure out a path

that doesn't box them in before collecting all of the items. This game and its Puzzler code implementation are showing in Figures E and F respectively.

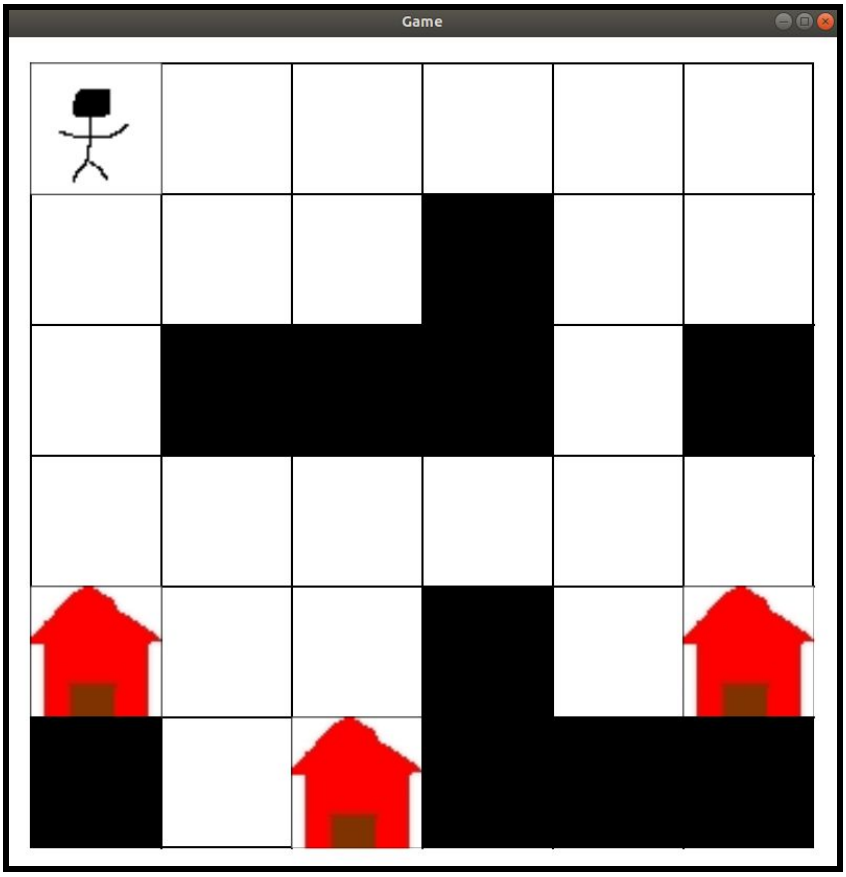

Figure E: A game about collecting the red items while not boxing the player in

```
#lang puzzler

START_MAP
P#####
###W##
#WWW#W
######
T##W#T
W#TWWW
END_MAP

draw:
  "P" -> "player.png"
  "W" -> "rect"
  "T" -> "coop.jpg"

action:
  "P": "up" -> (0, 1)
  "P": "down" -> (0, -1)
  "P": "left" -> (-1, 0)
  "P": "right" -> (1, 0)

interactions:
  "P" stop "W"
  "W" stop "W"
  "P" grab "T"

events:
  "P" onexit "W"

win:
  "T" count_items 0
```

Figure F: Puzzler code for the game shown in Figure E

The fourth and final game I created over the course of designing Puzzler is a game about sneaking to a specific spot on the map. The game, shown in Figure G, consists of the player (the black stick figure) moving on the grid and trying to not get in the "line of sight" as the guards (the red stick figures).

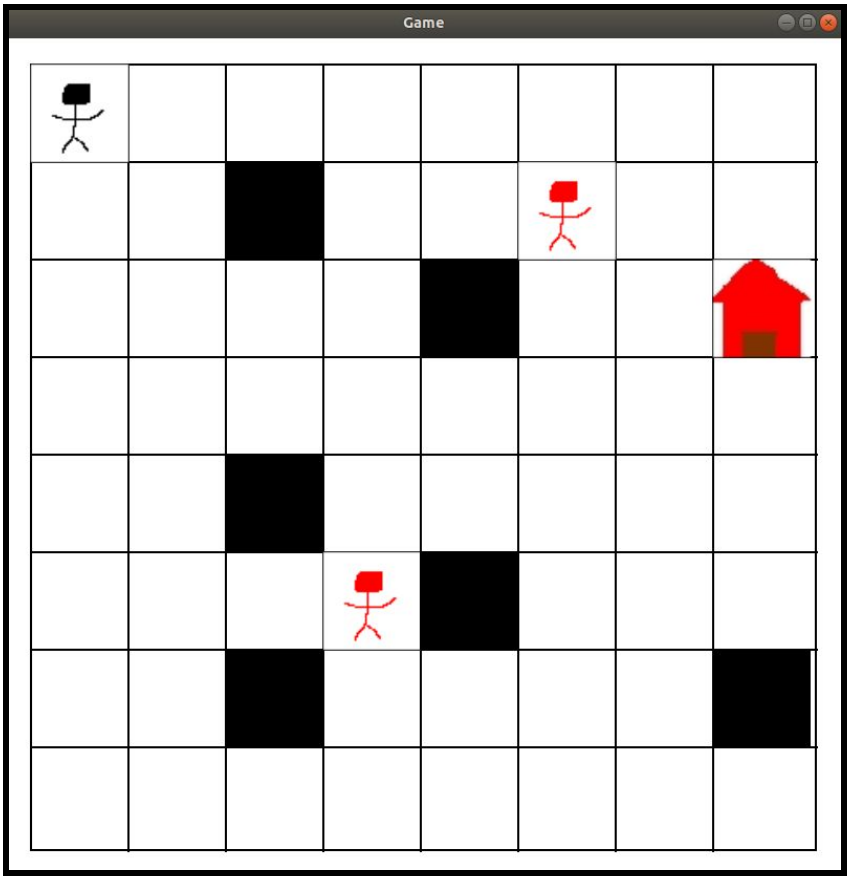

Figure G: A game about sneaking past guards

If the player gets in the same row or column with a guard and there are no blocks between them, the player loses the game and must restart. To win the game, the player has to push the black blocks in such a way that the guards cannot see the player as they makes their way to the red house. The Puzzler code for this game is shown in Figure H.

```
#lang puzzler

START_MAP
P#######
##B##G##
####B##T
########
##B#####
###GB###
##B####B
########

P#######
########
#######T
########
####B###
#######G
########
########
END_MAP

draw:
  "P" -> "player.png"
  "B" -> "rect"
  "T" -> "coop.jpg"
  "G" -> "guard.png"

action:
  "P": "up" -> (0, 1)
  "P": "down" -> (0, -1)
  "P": "left" -> (-1, 0)
  "P": "right" -> (1, 0)

interactions:
  "P" push "B"
  "P" grab "T"
  "B" stop "G"

lose:
  "G" straight_path_to "P"

win:
  "T" count_items 0
```

Figure H: The Puzzler code for the sneaking game in Fig. G

VI. Conclusions

The Puzzler language can be used to express many different 2D puzzle games in a very concise and clean way. When compared to the C++ alternatives, Puzzler can be used to represent equivalent 2D puzzle games in a fraction of the total source code size

and, more importantly, complexity of the system. Puzzler programs are written in a much neater, declarative syntax that allows for more meaningful changes in much less time than more general purpose languages.

The Racket programming language was a good choice for implementing Puzzler. Racket allowed me to concisely define Puzzler's syntax using a Backus-Naur Form (BNF) grammar. Racket's powerful metaprogramming (macro) system allowed me to define Puzzler's semantics by writing macros that translate parse trees into executable Racket code. Links to the entire Racket source code for the Puzzler language are provided in the appendix. Overall, Puzzler is a beginner friendly tool and can be used to make many different kinds of 2D puzzle games.

VII. The Future of Puzzler

As the designer of Puzzler there are many improvements that I am focusing on for the future development of the language. The most important addition to Puzzler will be more interactions and win rules so that a wider variety of more complex games can be expressed in the language. The win and lose states are also a priority, as they do not allow any customization from the author of Puzzler programs. To improve these I am experimenting with a system to allow for configurable win and lose screens. One final feature to Puzzler that would increase the span of the language is custom interactions. For example, rather than using the built-in "push" interaction that I created, a user would write Puzzler code to describe what "push" means. This would allow Puzzler

writers to add as many interactions as they need for a given game, and it would drastically increase the power of the Puzzler language.

Appendix

Definition of Puzzler Terms

map section - a representation of one or multiple game grids using single letters to describe game entities

action section - a series of action rules, which describe key input in a Puzzler game

interactions section - a series of interaction rules, which define what happens when one entity moves into a block with another entity

draw section - a series of draw rules, which tell the Puzzler game what image on the user's computer a specific entity should be drawn with

win section - a series of win/lose rules, which describe the different win/lose conditions in a Puzzler game

goal map section - a type of win condition defined similar to the map section, these maps are "goals" for a level, so when the current map looks like the goal map, the player wins the game

Puzzler Source Code (all hosted on GitHub):

parser.rkt (https://github.com/aowens-21/puzzler/blob/master/parser.rkt)

- This is the Backus-Naur Form grammar which represents the Puzzler language. It is written in the "brag" language, which generates a parser for a grammar.

tokenizer.rkt

(https://github.com/aowens-21/puzzler/blob/master/tokenizer.rkt)

- This is a file which describes all of the lexical tokens in of the Puzzler programming language, which **parser.rkt** uses to build up the grammar.

main.rkt (https://github.com/aowens-21/puzzler/blob/master/main.rkt)

- The entry point for the Puzzler compilation process. Defines a reader module and passes in the parse tree from the reader to **expander.rkt** for code generation. This is not unique to Puzzler and is a common Racket language creation pattern.

expander.rkt

(https://github.com/aowens-21/puzzler/blob/master/expander.rkt)

- Takes in the parse tree for a Puzzler program and does code generation via Racket macros. Holds many macros and also information about the state of a Puzzler game.

game.rkt

(https://github.com/aowens-21/puzzler/blob/master/expander-helpers/game.rkt)

- Defines a custom Racket class to represent a Puzzler game as well as many associated methods to describe behavior. One component used by the **expander.rkt** file to manipulate game state.

renderer.rkt

(https://github.com/aowens-21/puzzler/blob/master/expander-helpers/renderer.rkt)

- Defines a custom Racket class to handle all drawing and rendering for a Puzzler game. This piece is "plugged in" to **game.rkt** to handle all visual representation of the Puzzler game.

free_the_chick_cpp

(https://github.com/aowens-21/free_the_chick_cpp)

- This is a github repository containing the source code of the "Free the Chick" game, but implemented in C++, a more mainstream game programming language. This provides a good basis for comparison to the Puzzler implementation and its benefits.

Works Cited

PuzzleScript - an open-source HTML5 puzzle game engine. Stephen Lavelle, n.d., puzzlescript.net/index.html. Accessed 14 April 2019.

Midway Games. *Midway Galaga Parts and Operating Manual*. Chicago, Illinois, Midway Games, 1981.

Valve Software. "*ValveSoftware/halflife: Half-Life 1 engine based games*". *GitHub*. https://github.com/ValveSoftware/halflife. Accessed 14 April 2019.