Mouse vs. Machine: The Game

Cafferty Aiko Frattarelli
Bard College, cf4707@bard.edu

Follow this and additional works at: https://digitalcommons.bard.edu/senproj_s2017

Part of the Artificial Intelligence and Robotics Commons, Digital Humanities Commons, and the Graphics and Human Computer Interfaces Commons

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 4.0 License.

Recommended Citation

This Open Access work is protected by copyright and/or related rights. It has been provided to you by Bard College's Stevenson Library with permission from the rights-holder(s). You are free to use this work in any way that is permitted by the copyright and related rights. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself. For more information, please contact digitalcommons@bard.edu.
Mouse Versus Machine: the Game

Senior Project submitted to
The Division of Science, Mathematics, and Computing
of Bard College

by

Cafferty Aiko Frattarelli

Annandale-on-Hudson, New York
May 2017
Acknowledgements

I would like to thank my senior project advisor, Khondaker Salehin, for his help making this project a reality, and my academic advisor, Keith O’Hara for his help and encouragement throughout my school career.

Particular thanks to Benjamin Newman, who provided many useful tools and another set of eyes for my code, and William Flack, who let me use his dorm as a spare room to work in, and for the lunches we all spent together getting much needed relaxation.

And to my friends who I worked alongside to get our projects done, Nicolas Engst Matthews, who I must also thank for proofreading my writing, Hannah Livant, Haley Goss-Holmes, and Hayden Zahn. I am terrifically grateful for your company.

And to all my friends throughout college, making these years to remember.

And to my family, without whom I wouldn’t be here at all.
Abstract

Many modern video games built by big name companies are coded by a group of people together using, and possibly modifying, an already designed game engine. These games usually have another group of people creating the artwork. In this project, I coded and designed a video game from scratch, as well as created all the artwork used in the game. The player controls a mouse character who fights a variety of monsters. In order to create the complexity of the game, I implement basic neural networks as the enemy artificial intelligence, i.e. the decision making process of the enemy. It uses this to learn how to combat a player from the player’s actions, including movement and attacking. Movement is implemented through changing the player’s position on the screen, and attacking creates an image which causes damage to other characters. The program is coded in Python, using the Pygame library for displaying graphics. It is currently an alpha version, with the code built and all the gameplay elements in place. With the existing foundation, this game, “Mouse versus Machine”, can be extended into a full-fledged game in the future.
# Table of Contents

Acknowledgements ........................................................................................................... i

Abstract .......................................................................................................................... iii

Table of Contents ............................................................................................................... v

1 - Introduction .................................................................................................................. 1

2 - An Overview of the Code ............................................................................................. 3
   2.1 - Sprite class ............................................................................................................. 4
   2.2 - Character class ..................................................................................................... 6
   2.3 - Enemy class ......................................................................................................... 8
   2.4 - Inanimate class .................................................................................................... 10
   2.5 - Attack class ....................................................................................................... 10
   2.6 - Door class .......................................................................................................... 11
   2.7 - Words class ....................................................................................................... 12
   2.8 - Main function .................................................................................................... 12

3 - Neural Networks ........................................................................................................ 15
   3.1 - The Basics .......................................................................................................... 15
   3.2 - The Code ............................................................................................................ 16
      3.2.1 - The Variables .............................................................................................. 17
      3.2.2 - The Functions ............................................................................................ 18

4 - Art Design and Progression ....................................................................................... 21

5 - The Process ................................................................................................................ 27
   5.1 - The Coding ......................................................................................................... 27
   5.2 - The Bugs (and other technical difficulties) ......................................................... 29

6 - Conclusion .................................................................................................................. 31

7 - Appendix .................................................................................................................... 33
   7.1 - Main Code ......................................................................................................... 33
   7.2 - Test Functions ................................................................................................. 51
   7.3 - Survey .............................................................................................................. 54

8 - Bibliography .............................................................................................................. 55
1

Introduction

This project started with the concept of building a game from scratch. The goal was to learn about the building blocks of video games and how those can be expanded upon, as well as using these building blocks as a venue for a decision-making artificial intelligence (A.I.). The process involved research into game design and A.I. development. I wanted to use minimal outside libraries aside from one for graphics to display the game. I also wanted to incorporate different aspects of my college experience, including my computer science experiences, particularly in Object Oriented Programming, and Intelligence and Perception in Robotics, as well as my art based experiences - mainly Cybergraphics - though with a basis in the various other art classes I have taken, as well as my job on campus as a poster designer.

Game design is a complex subject involving many facets of computer programming, as well as art and design. Since I am only one person, I could not hope to rival the works of major video game companies and various designers using video game production as their sole form of
employment. This limitation caused me to focus on creating the building blocks of the game and making my own program that I could work with and continue building into a completed game. This would also allow me to have a strong understanding of what was and was not possible with the game, and why. The goal was to have complete control over the game design and structure, as well as working to optimize the game processes for minimum lag overall.

The game design was influenced by many sources, including my own experience with video games, reading various articles about video games, and discussing video game design and game mechanics with my peers. I ended up going with a relatively common overall structure, with a top-down viewpoint of a dungeon where the player controls one character and progresses through a series of rooms, fighting different enemies along the way. Some games that are comparable to this are the old Legend of Zelda games for the GameBoy, and the game Binding of Isaac, a more recent game for the PC. These are both variations of top-down two-dimensional games. Two-dimensional games are generally either top-down or side-scrolling. Viewpoints, and graphics in general, get more complicated and computer intensive once it gets to three-dimensional representations, so I kept this game two-dimensional.

The main goals behind using a neural network for the A.I. in this game was to have an amount of unpredictability and reactability in the enemy characters’ actions. The goal was not to have a perfect A.I., because that does not lead to particularly entertaining gameplay. Since this neural network starts with random weights and gathers data as the game progresses, it fulfills both of the criteria of being unpredictable and reactive.
2 An Overview of the Code

This game was coded in Python 2.7.11, implementing the Pygame and NumPy modules. Pygame is a Python library used for building video games. In this project it was used for its graphics and game time functions. The NumPy module was used for all the mathematical functions that weren’t basic arithmetic, including basic trigonometric math to calculate in game distances, as well as array multiplication for the neural network, and random number generation for both the neural network and determining how the enemy characters walk when they are not aimed towards a specific location in the game.

Classes

Since this game was programmed using object-oriented programming, it is made up of several class objects, which are listed and described below, along with their initial variables and
functions; not including the basic functions that only set different variables. The code is listed in the appendix for reference. Notice the NeuralNetworkAI class is not assessed here. This is because it is more complex and will be covered thoroughly in the next section.

2.1 The Sprite class

This class is the basic class used to display any sort of image that may need to be interacted with or moved. It is the parent class of every class in the program other than the NeuralNetworkAI, since every other class represents some in-game object.

Variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“position”</td>
<td>a tuple holding the x and y coordinates of the Sprite</td>
</tr>
<tr>
<td>“size”</td>
<td>a tuple holding the width and height of the Sprite</td>
</tr>
<tr>
<td>“image”</td>
<td>loads and stores the image for the Sprite</td>
</tr>
<tr>
<td>“sprite”</td>
<td>stores the Sprite’s display image, which can be changed, for when multiple images are possible</td>
</tr>
<tr>
<td>“interactable”</td>
<td>a Boolean that says whether a Sprite can be interacted with (i.e. there is a response if the player presses the interact key while their character is looking at the Sprite)</td>
</tr>
<tr>
<td>“text”</td>
<td>a tuple of strings that display when the object is interacted with</td>
</tr>
<tr>
<td>“substantial”</td>
<td>Boolean that determines whether an object can be walked through or not</td>
</tr>
<tr>
<td>“flammable”</td>
<td>Boolean that determines if an object is hurt by the player’s attack</td>
</tr>
<tr>
<td>“enemy”</td>
<td>Boolean that determines if the object is an enemy character</td>
</tr>
<tr>
<td>“direction”</td>
<td>an integer value between -3 and 4 that determines which direction the Sprite is facing</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

**The Main Functions:**

<table>
<thead>
<tr>
<th>Place</th>
<th>takes a position and a surface and uses Pygame’s blit function to display the object on the surface</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoesOverlap</td>
<td>checks if the current object overlaps with another object, provide to the function, by comparing the two objects’ position and size, used in various collision detections</td>
</tr>
<tr>
<td>OutOfBounds</td>
<td>checks if object goes off the screen</td>
</tr>
<tr>
<td>CanInteract</td>
<td>takes another object and sees if the current object (usually the player character) can interact with the other object, taking into account if the other object is “interactable” and the distance between the current object and the other object. It tests this by seeing if the player’s sprite would overlap, using the DoesOverlap function, with the object if moved forward by half of the Sprite’s size.</td>
</tr>
<tr>
<td>SetText</td>
<td>sets the text to the given tuple of strings and sets the Sprite’s “interactable” variable to the given Boolean, “interact”, defaulting to True, since generally if the text is set, the object can be interacted with to at least provide a text response. Will be set to False if removing the text.</td>
</tr>
<tr>
<td>Delta</td>
<td>converts a given direction and distance into change in x and y, returned as a tuple</td>
</tr>
</tbody>
</table>
**DistanceTo** finds the distance from approximately the edge of one object to the edge of the other, using the size of the object to find the center as well as finding the approximate distance from the center to the edge.

**SetDirection** sets “direction” to the given direction. If given an integer outside of integer values between -3 and 4, the function converts it to inside the range. *SetDirection* also changes the sprite displayed to reflect the set direction.

### 2.2 The Character class

The Character class is a child class of the Sprite class. It is the base class for all characters, mainly the player character and the enemy monsters.

**Variables:**

<p>| “attributes” | a list with four elements, corresponding to the Character’s health, attack, defense, and speed, in that order. Health determines how many hits a Character can take before it dies. Attack determines how often a Character can attack. Defense gives the odds of an attack hitting the Character. Speed determines how far a Character can go each game tick. |
| “canAttack” | if “canAttack” is zero, the Character can attack, otherwise “canAttack” is an integer above zero that is manipulated by the <em>IncrementAttack</em> function to reach zero in a time frame based on the Character’s attack attribute. |
| “level” | an integer that determines the difficulty of the enemies the Character, applicable to the player character. |</p>
<table>
<thead>
<tr>
<th><strong>“stepNum”</strong></th>
<th>an integer that keeps track of the number of ticks the Character has walked, so the image can change depending on the number of steps and looks like it’s walking.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>“step1”</strong></td>
<td>stores the loaded and scaled image for the first part of the walking animation.</td>
</tr>
<tr>
<td><strong>“step2”</strong></td>
<td>stores the loaded and scaled image for the second part of the walking animation.</td>
</tr>
<tr>
<td><strong>“attack”</strong></td>
<td>creates and stores an Attack object with a default attack image.</td>
</tr>
</tbody>
</table>

**The Main Functions:**

| **GetHealth, GetAttack, GetDefense, and GetSpeed** | returns the value of each respective attributes from the “attributes” list |
| **AdjustAttribute** | takes a string “Health”, “Attack”, “Defense”, or “Speed” and adds the given change value to the attribute to adjust it. The change value can be negative. |
| **CanAttack** | returns True if “canAttack” is zero, and sets “canAttack” to a higher integer based on the Character’s attack attribute, otherwise it returns False. |
| **IncrementAttack** | decrements the “canAttack” value by one each game tick until it reaches zero. |
| **Walk** | moves Character in the given direction based on the speed of the Character by calling IncrementalWalk, as well as updating the displayed image for the character based on “stepNum”. Returns the value from IncrementalWalk. |
**IncrementalWalk**

tries to move the Character in a given direction and distance by checking if the Character can be placed that distance and direction from where it starts using *DoesOverlap* and *OutOfBounds*. If it can be placed there, it is placed there, otherwise it calls *IncrementalWalk* again with a lower distance, until the distance reaches zero, in which case it returns False. If a successful move is accomplished it returns True.

**Attack**

displays the Character’s attack sprite when necessary and if the Character can attack, it checks if the attack sprite overlaps with any object that can be damaged by the attack, and if so it decreases the health of said object, and if that object’s health reaches zero it removes the object from the objects array given. Also changes the success value of the NeuralNetworkAI.

### 2.3 The Enemy Class

The Enemy class is a child of the Character class, and by extension the Sprite class. It holds the variables for several different Enemy monsters with the potential to add more quite easily. It also has the functions that allow the Enemy to use the programmed A.I. to make decisions and control its actions.
Variables:

<table>
<thead>
<tr>
<th>“type”</th>
<th>a string variable that determines what kind of monster the Enemy is. Currently the options are “spider”, “lizard”, and “other”. Each has a different image and different attributes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>“move”</td>
<td>stores the move decided by the neural network A.I., an array of two integers, zero or one.</td>
</tr>
<tr>
<td>“wander”</td>
<td>this variable is an integer used to count how long the Enemy has been using the <em>Wander</em> function and not changed direction.</td>
</tr>
<tr>
<td>“status”</td>
<td>keeps track of whether the Enemy can make it’s next move yet. Used to give a time where it has to keep using a single move so it cannot react immediately and gives the player Character a chance to act.</td>
</tr>
<tr>
<td>“action”</td>
<td>stores the current action the Enemy is performing or has just performed.</td>
</tr>
<tr>
<td>“lastAction”</td>
<td>saves the last action the Enemy performed once the current action changes.</td>
</tr>
<tr>
<td>“previousHealth”</td>
<td>saves the health the Enemy had last game tick.</td>
</tr>
<tr>
<td>“usingAI”</td>
<td>determines if the Enemy is using the neural network A.I. or the simple state-based A.I. It’s True if using the neural network and False otherwise.</td>
</tr>
</tbody>
</table>

The Main Functions:

<table>
<thead>
<tr>
<th>Act</th>
<th>calls <em>ChooseAction</em> and executes the provided “action” by name.</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ChooseAction</em></td>
<td>uses different procedures depending if the Enemy is using the neural network A.I. or not. If it is, <em>ChooseAction</em> creates a new data point and evaluates it with the neural network and returns the result as the action to perform. Otherwise, <em>ChooseAction</em> uses the Enemy’s distance from the player and</td>
</tr>
</tbody>
</table>
direction towards the player to determine its next action.

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Combat</strong></td>
<td>the “action” called from <code>Act</code> that calls <code>Attack</code>, defined in the Character class.</td>
</tr>
<tr>
<td><strong>Flee</strong></td>
<td>the “action” called from <code>Act</code> that moves the Enemy the opposite direction from the player.</td>
</tr>
<tr>
<td><strong>Turn</strong></td>
<td>the “action” called from <code>Act</code> that moves the Enemy towards the player, and attempts to go around obstacles.</td>
</tr>
<tr>
<td><strong>Wander</strong></td>
<td>the “action” called from <code>Act</code> that has the Enemy <code>Walk</code> in a random direction.</td>
</tr>
</tbody>
</table>

### 2.4 The Inanimate Class

The Inanimate class is a child of the Sprite class. It is used to display inanimate objects such as walls and rocks. It also suggests the structure for objects that can be broken.

**Variables:**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“sturdy”</td>
<td>how much the Inanimate can take before being “broken”</td>
</tr>
<tr>
<td>“broken”</td>
<td>if the Inanimate can be used</td>
</tr>
</tbody>
</table>

### 2.5 The Attack Class

The Attack class is a child of the Sprite class. It is used to display attacks.

**Variables:**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“hold”</td>
<td>determines if the Attack image displays even if the Attack is not doing damage</td>
</tr>
<tr>
<td>“image1” and</td>
<td>the image displayed for the Attack and the flipped image for animating the</td>
</tr>
</tbody>
</table>
"image2" | Attack
---|---
"state" | keeps track of how long part of the Attack animation has been showing so it can switch back and forth to animate it.

The Main Functions:

| Show | displays and animates the Attack image |

### 2.6 The Door Class

The Door class is a child of the Sprite class. If the player interacts with the Door, they will be moved from one room to another, unless the Door is “locked.” The Door class defaults to “interactable” being True.

**Variables:**

| “nextRoom” | the room, a collection of objects, to which the Door opens |
| “locked” | determines whether the Door can be opened or not. If it is “locked” it cannot be opened until otherwise determined. Defaults to being not “locked”. The Door’s “text” is determined by “locked” as well. |
| “opensTo” | the location the Door opens to, i.e. where the player is placed if they interact with the Door and it is not “locked”. |

The Main Functions:

| Unlock | sets “locked” to False |
| Open | returns “nextRoom” and “opensTo” to be used to move the player Character to the intended location through the Door |
2.7 The Words Class

The Words class is a child of the Sprite class. It is used to display all text in the game.

Variables:

<table>
<thead>
<tr>
<th>“size”</th>
<th>overwrites the Sprite “size” variable to one to be used for font size</th>
</tr>
</thead>
<tbody>
<tr>
<td>“text”</td>
<td>stores the text to be displayed in a tuple of various lengths</td>
</tr>
<tr>
<td>“font”</td>
<td>loads the font to be used for the text</td>
</tr>
</tbody>
</table>

The Main Functions:

<table>
<thead>
<tr>
<th>Place</th>
<th>overwrites Sprites’ Place function to render each line of the Word’s “text” in its “position”, iterating through the “text” to put each line in the Word below the previous line. Also slightly lightens the Word’s color with each line, so large chunks of text get lighter towards the bottom.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetText</td>
<td>used to set the Word’s “text” variable so if a chunk of text is added to the Word it goes to the top of the stack and excess lines are removed from the bottom. Mainly used for the text display from interacting with objects in the game.</td>
</tr>
</tbody>
</table>

2.8 The Main Function

The main() function is the function outside of the classes that defines the game loop and variables carried over from one game session to another. It brings together all the different classes and functions defined above and runs the game.
It starts by initiating all the constants and objects with the desired parameters, including the game clock, window for displaying images on, the player, monsters, and a variety of inanimate objects. These objects are placed in the different room lists, which can be swapped between being used in the current OBJECTS list, which determines which objects are being displayed.

Then there is the game loop, which is a while loop that continue indefinitely until the player exits the game. The game acts different if the player Character is alive or dead. If the player is dead, meaning their health has reached zero, the game will display the death screen and give the player the option to restart the game. If the player is not dead, the game loop continues normally, logging the player’s keystrokes. The “w”, “a”, “s”, and “d” keys correspond to moving the player Character up, left, down, and right respectively, while the left shift key has the player Character try to interact with an object, and the space bar causes the player Character to attack. After these are checked, the objects in the current room are assessed to see if they should do anything, such as having enemies move and attack. Then the object images are redrawn in the window. Finally, health and other statuses of the player Character are updated and one game tick passes.

Below the main function several constants are defined, including the display size, and the input and output arrays for initializing the neural network A.I., and the main function is called.
3

Neural Networks

3.1 The Basics

Neural networks are based on neurons and the networks they create. In the case of computers, these neurons are an approximation: computational structures that take inputs and return activation values. Each neuron has a weight, learned through a training dataset, that is applied to the input, then run through a nonlinear function, such as sigmoid or hyperbolic tangent to find the activation value. When these neurons are connected together to process the data through hidden layers they can process a large variety of complex data sets.
Here is an example of a basic neural network with two inputs, one hidden layer with three neurons, and two outputs.

### 3.2 The Code

The neural network in this game is implemented through the class `NeuralNetworkAI`, which collects and analyzes data from every Enemy when activated. Data collection is accomplished by retrieving the values already stored in the Enemy class, or through calling a function in the Enemy class. Analyzing the data is accomplished through forward propagation and backpropagation. Forward propagation means to multiply the input through the network to get an output. Backpropagation involves finding the amount of error between the predicted result
from forward propagation and the given result and using that to update the weights for more accurate prediction.

### 3.2.1 The Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“totalData”</td>
<td>the total number of sample data there is to use, calculated by the length of the given “initialInputData”.</td>
</tr>
<tr>
<td>“dataLength”</td>
<td>the number of inputs per each piece of data, plus one to accommodate the bias, calculated using the length of one of the inputs</td>
</tr>
<tr>
<td>“weights1, 2, and 3”</td>
<td>randomly generated sets of weights for each layer of the neural network, to be changed to reflect the data through backpropagation</td>
</tr>
<tr>
<td>“inputData”</td>
<td>previously generated input data to be analyzed</td>
</tr>
<tr>
<td>“outputData”</td>
<td>previously generated output data corresponding to the input data to be analyzed</td>
</tr>
<tr>
<td>“unclaimedData”</td>
<td>a variable to store input data that is being collected but has yet to be analyzed</td>
</tr>
<tr>
<td>“unclaimedResults”</td>
<td>a variable to store output data that is being collected but has yet to be analyzed, corresponds to “unclaimedData”</td>
</tr>
<tr>
<td>“currentSuccessValue”</td>
<td>stores the current success value of the “unclaimedData” and “unclaimedResults”, a calculation of the Enemy’s action’s success</td>
</tr>
<tr>
<td>“learningRate”</td>
<td>the rate at which each update to weights affects the weights, useful for keeping the weights from converging prematurely</td>
</tr>
</tbody>
</table>
3.2.2 The Functions

-trainNeuralNetwork

This function first adds an extra 1 to the end of each array of input data to be used for the bias. Then it updates the weights the given number of times using backpropagation. This neural networks has two hidden layers.

-newDataPoint

This function collects the current direction, distance and direction to the player, edited to be a value between 0 and 1, from one Enemy and returns it to whichever function was looking for that information.

-collectData

This function collects a data point from newDataPoint and places it into “unclaimedData”, as well as collecting the current action from the Enemy and storing it in “unclaimedResults”. It also takes a variable “Refresh”, which defaults to False, that determines if collectData will attempt to see if the “currentSuccessValue” is high enough to add the unclaimed data and results to “inputData”. When adding the new data to “inputData”, it must remain the same length in order for the matrix multiplication works, so the function shuffles the new data and the old data together and takes a random sample of the two so there are the same number of
inputs in the updated array. If it does add new data, it calls the *trainNeuralNetwork* function again, with a slightly higher learning rate.

- **extrapolate**

  This function takes the current state of the Enemy and uses forward propagation with the current weights to choose the next action the Enemy takes.

- **sigmoid**

  This function converts the outputs from each layer nonlinearly to a number between 0 and 1, as well as being able to be used to find the derivative. The derivative is used to find how certain the predicted value was. If the derivative, and thus the slope, is high, then it is not a very certain prediction, so the change in weight will be more drastic when the weights are updated.
4

Art Design and Progression

The art of the game and how it progressed through the game design

First sketches of possible characters for the player character. The mouse was eventually chosen to keep the scale reasonable and the complexity down.
Sketches done of possible game scenarios and enemy designs. Several of these can be implemented using the data structures and objects created.
Original digital main character mouse design, scrapped due to decision to have the game view from the top down.

Second mouse design, scrapped due to the logistical difficulties caused by a rectangular sprite design. Mainly the problems occurred in corners and near objects where it would either clip through them or not fit in places where it fit before when it was sideways.
Final mouse design, showing differences in walking animation.

The sprite for the spider and the lizard-like enemy characters.

The sprite for the rock object, also used for the morph enemy as an enemy to catch the player off-guard.
The attack sprites used by the player and the enemy characters respectively.

The wall and door sprites. They define the dimensions of the room and show the way out.

Initial mock-up of gameplay.
The final background image, with spaces for displaying health, level, and enemies killed, as well as any in-game description or dialogue.

An example of a game scene where the player character is being attacked by a morph.
5

The Process

5.1 The Coding

Coding this game from the ground up, I started by figuring out which programming language to use. I settled on Python because it was the language I was most familiar with, particularly in regards to object oriented programming and in relation to artificial intelligence similar to what was covered in the Intelligence and Perception in Robotics class. I decided to use the Pygame module to supplement Python due to its graphics and timing library.

Next I started coding the rough building blocks of the game, fleshing out which classes were and were not necessary. In this step there was a lot of writing and re-writing code for efficiency, so the code did not work unnecessarily hard and cause the game to slow down. This
would cause frustration both on the player’s level and further down the line cause difficulties with training the A.I., which would only serve to slow the program further.

Once the code was to a point where I could introduce graphics, I began to digitally draw out the characters using my drawing tablet and the art program GIMP 2. Some of them had already been designed on paper, as seen in the previous section. After several redesigns I found what worked best for the game and drew up final versions.

Finally, I started implementing the A.I. At first the goal was to have the A.I. fully control the Enemy characters, that, however, turned out not to be feasible for a number of reasons. First, it slowed down the program a significant amount due to the number of relevant inputs, as well as the number of test problems that would be necessary to give the A.I. accurate instruction. Second, the problem with giving the A.I. full control of the Enemy characters was that the computer has much quicker reactions than a human player does, so if it did work fully, it would be incredibly difficult, if not impossible, to play the game and win, or even have fun. Thus, I decided on a partially hybridized A.I., where the neural network chooses between a few different actions with a time delay.

Implementing the neural network involved a variety of different tests, including ones to generate a training data set, as well as finding the right balance of number of iterations and learning rate for the neural network to produce accurate results.
5.2 The Bugs (and other technical difficulties)

There are always some bugs and other difficulties in any coding project, however these were some that were particularly difficult or stood out in some way, and some which still persist.

The Stuck Mouse

This was the bug where the player Character, the mouse, could not move because it was seeing itself as an obstacle to where it could walk. This was a result of a previous bug fix where the Enemy characters could walk through the player Character due to not perceiving it as an obstacle. This was eventually solved by reformatting the \textit{Walk} function so it could check whether the object it was trying to walk through was itself.

There was also a similar bug where, if the player Character was facing the wrong direction, its attack could hit itself and burn it to death, which was also solved by making it check whether the object it was attack was itself.

The Wall Approach Problem

This was a problem where any of the Characters did not always fully approach the wall while walking due to the Character only being able to move its full distance or not at all. This was solved by creating the function \textit{IncrementalWalk}, which calls itself recursively until it’s certain there’s no space to move into between the Character and the other object.
Beta testing

It was a goal to have some people beta test this game and fill out a survey based on their experiences. However, sending the game to others proved challenging in the time left, and when I did send it to some people. They proved to be unable to run the game on their computers due to various factors, such as what software they had on their computer, and what kind of computer they had. For example, one person had one version of Python downloaded that was not compatible with my version of Python, Also, Mac computers were particularly troublesome because of their differing file system. Thus I was unable to have the game beta tested as of this report. Some more research into creating an executable file of the game will likely make it possible in the future.

Diagonal Rotation

One problem that still remains in the code is the problem where rotating an image diagonally makes said image larger, and thus slightly changes how the image interacts with obstacles, sometimes causing the images to overlap. It is not a major issue, however it is still an issue I will continue to investigate. It wouldn’t be as much of a problem if Pygame had an image cropping function, however it does not so it requires a bit of a work around, and will most likely involve rewriting how the Place function works.
6

Conclusion

This game is the alpha version of the game, which I intend to complete down the line by adding more assets and features. Some features I will implement include other friendly characters that the player can interact with, as well as more varied objects and new monsters to fight in new rooms. Once these features are implement I can move onto expanding the game into a full story, possibly with sound and dialogue.

On the more technical side of things, I intend to make building objects and storing the training array for the neural network less cumbersome by reading it from a file instead of having all the values directly in the code. I will also look into an improved training set for the neural network, with a more sophisticated use of the success value that was implemented in this version
- possibly storing the success value of each input. This will involve some research into effectively calculating success in the game scenario. Possible contributing factors I have considered are taking less damage from the player, dealing more damage to the player, how long the Enemy character survives, and a ratio of the damage done over time versus the damage taken. The most complex aspects of figuring this out will be balancing the Enemy’s survival with defeating the player character, as well as how to calculate these values and over what amount of time.

Overall, however, this project was successful in what it set out to do, namely, creating an program where the basics of the game were implemented and the actions of the Enemy characters depended on the neural network, which I coded. I have come out of this project with a much more solid understanding of how neural networks work, as well as a product that I am proud of. Despite some errors and frustrations in the process, at the end the work I did over the past year it came together into a successful program.
7

Appendices

7.1 Main Code

1. #Cafferty Frattarelli
2. #Mouse Vs. Machine: the Game
3. #Senior Project - May 2017
4. 
5. import pygame, sys
6. from pygame.locals import *
7. import numpy
8. 
9. #Sprite class - base class for all game objects
10. class Sprite:
11.     def __init__(self, sprite, size = (100, 100)):
12.         self.position = (0,0)
13.         self.size = size
14.         self.image = pygame.image.load(sprite)
15.         self.image = pygame.transform.scale(self.image, (self.size[0],
16.                         self.size[1]))
17.         self.sprite = self.image
18.         self.interactable = False
19.         self.text = " "
20.         self.substantial = True
21.         self.flammable = False
22.         self.enemy = False
23.         self.direction = -2
24.         #Up=0, UpRight=-1, Right=-2, DownRight=-3, UpLeft=1, Left=2,
25.         DownLeft=3, Down=4
26.         #Displays the sprite in assigned position (takes position tuple (x,y) and
27.         #surface to put it on)
28.         def Place(self, position, surface):
29.             surface.blit(self.sprite, position)
30.             if self.GetPos() != position:
31.                 self.SetPos(position)
32.         #Returns current position
33.         def GetPos(self):
34.             return self.position
35.         #Sets current position (takes a tuple with two values, x and y)
36.         def SetPos(self,position):
37.             self.position = position
38.         #Returns Sprite size (returns a tuple with two values, x and y)
39.         def GetSize(self):
40.             return self.size
#Sets Sprite size (takes a tuple with two values, x and y)
```
def SetSize(self, size):
    self.size = size
    self.sprite = pygame.transform.scale(self.image, (self.size[0],
        self.size[1]))
```

#Returns if something can be walked through or not (returns True or False)
```
def GetSubstantial(self):
    return self.substantial
```

#Sets if something can be walked through or not (takes True or False)
```
def SetSubstantial(self, sub):
    self.substantial = sub
```

#Sees if current object overlaps other object, used for collision detection
(takes and Sprite object, returns True or False)
```
def DoesOverlap(self, other):
    if other.GetSubstantial():
        selfx = self.GetPos()[0]
        selfx2 = selfx + self.GetSize()[0]
        selfy = self.GetPos()[1]
        selfy2 = selfy + self.GetSize()[1]
        otherx = other.GetPos()[0]
        otherx2 = otherx + other.GetSize()[0]
        othery = other.GetPos()[1]
        othery2 = othery + other.GetSize()[1]
        if selfx >= otherx2 or selfy >= othery2 or selfx2 <= otherx or selfy2
            <= othery:
            return False
        else:
            return True
    else:
        return False
```

#Used to make sure we haven't placed an object off screen (returns True or False)
```
def OutOfBounds(self):
    selfx = self.GetPos()[0]
    selfx2 = selfx + self.GetSize()[0]
    selfy = self.GetPos()[1]
    selfy2 = selfy + self.GetSize()[1]
    if selfx < 0 or selfx < 0 or selfx2 > DISPLAY_X or selfy2 > DISPLAY_Y:
        return True
    else:
        return False
```

#Checks if something is interactable (returns True or False)
```
def Interactable(self):
    return self.interactable
```

#Sets if something is interactable (takes True or False)
```
def SetInteractable(self, interact):
    self.interactable = interact
```

#Checks if this object can interact with supplied object,
given it can interact with something up to it's own size in front of it
(takes another Sprite object, returns True or False)
```
def CanInteract(self, other):
    direction = self.GetDirection()
    delta = self.Delta(direction, self.GetSize()[0], self.GetSize()[1])
    tempPos = self.GetPos()
    self.SetPos((tempPos[0] + delta[0], tempPos[1] + delta[1]))
    result = self.DoesOverlap(other)
    self.SetPos(tempPos)
    if other.Interactable() and self != other:
        return result
    else:
        return False

#Sets text that displays when this object is interacted with, and sets the
#object as interactable
#(takes a tuple of strings)
def SetText(self, text, interact=True):
    self.text = text
    self.interactable = interact

#Returns interaction text (returns a tuple of strings)
def GetText(self):
    return self.text

#Converts direction to change in x and y (returns a tuple with two values,
x and y)
def Delta(self, direction, distance1, distance2=-1):
    if distance2 == -1:
        distance2 = distance1
    delta = (distance1 * numpy.cos(numpy.pi * direction / 4),
             distance2 * numpy.sin(numpy.pi * direction / 4))
    return delta

#Finds distance to another object from center minus half of size
def DistanceTo(self, other):
    here = (self.GetPos()[0] + self.GetSize()[0] / 2,
    there = (other.GetPos()[0] + other.GetSize()[0] / 2,
              other.GetPos()[1] + other.GetSize()[1] / 2)
    return numpy.sqrt((there[0] - here[0]) ** 2 + (there[1] - here[1]) ** 2 - (self.size[0] + self.size[1] +
                                                                        other.size[0] + other.size[1]) / 4)

#Returns direction from this to other object (takes a Sprite object and
returns direction as defined in Sprite.__init__() )
def DirectionTo(self, other):
    xDiff = self.GetPos()[0] - other.GetPos()[0]
    yDiff = self.GetPos()[1] - other.GetPos()[1]
    direction = round(4 + numpy.arctan2(yDiff, xDiff) / numpy.pi * 4)
    while direction > 4:
        direction = direction - 8
    while direction < -3:
        direction = direction + 8
    return direction

#Returns direction (an integer)
def GetDirection(self):
    return self.direction

#Sets direction to an integer between -3 and 4 so it’s useable by other
functions (takes an integer)
def SetDirection(self, direction):
    if direction >= -3 and direction <= 4:
        self.direction = direction
        self.sprite = pygame.transform.rotate(self.image, 45*(-direction-2))
    elif direction > 4:
        self.SetDirection(direction-8)
    elif direction < -3:
        self.SetDirection(direction+8)

# Character class - base class for all characters: monsters, player, etc.
class Character(Sprite):
    def __init__(self, img1, img2, size=(100,100)):
        Sprite.__init__(self, img1, size)
        self.attributes = [20,10,10,15]
        # Health, Attack, Defense, Speed
        self.flammable = True
        self.canAttack = 0
        self.level = 1
        self.stepNum = 0
        self.step1 = pygame.image.load(img1)
        self.step1 = pygame.transform.scale(self.step1, (self.size[0], self.size[1]))
        self.step2 = pygame.image.load(img2)
        self.step2 = pygame.transform.scale(self.step2, (self.size[0], self.size[1]))
        self.attack = Attack("../Art/Fire.png", True, (50,50))

    # returns character's current health (returns an integer)
    def GetHealth(self):
        return self.attributes[0]

    # returns character's current attack stat (returns an integer)
    def GetAttack(self):
        return self.attributes[1]

    # returns character's current defense stat (returns an integer)
    def GetDefense(self):
        return self.attributes[2]

    # returns character's current speed stat (returns an integer)
    def GetSpeed(self):
        return self.attributes[3]

    # takes a string with the attribute name and adds the change to the stat (change can be negative)
    def AdjustAttribute(self, att, change):
        if (att == "Health"):
            self.attributes[0] = self.attributes[0] + change
        elif (att == "Attack"):
        elif (att == "Defense"):
        elif (att == "Speed"):
        else:
            print("Error: No such attribute")

    # returns boolean of True if the character can attack and False if it cannot
def CanAttack(self):
    if self.canAttack == 0:
        self.canAttack = 13 - self.GetAttack()
        return True
    else:
        return False

    # counts down to when the character can attack again, based on the attack stat
    #(would more accurately be called attack speed, but is not for reasons of clarity)
    def IncrementAttack(self):
        if self.canAttack > 0:
            self.canAttack = self.canAttack - 1
        elif self.canAttack < 0:
            self.canAttack = 0

    # Moves character in given direction if possible, using IncrementalWalk
    def Walk(self, direction, objects):
        self.stepNum += 1
        speed = self.GetSpeed()
        if self.stepNum >= 2*speed/10:
            self.image = self.step1
            self.stepNum = 0
        elif self.stepNum >= speed/10:
            self.image = self.step2

        return self.IncrementalWalk(direction, objects, speed)

    # Moves character based on direction and speed, gets as close to an obstacle as possible, returns how far it went
    def IncrementalWalk(self, direction, objects, distance):
        self.SetDirection(direction)
        tempPos = self.GetPos()
        delta = self.Delta(direction, distance)
        self.SetPos((tempPos[0]+delta[0], tempPos[1]+delta[1]))

        if distance <= 0:
            self.SetPos(tempPos)
            return False
        if self.OutOfBounds():
            self.SetPos(tempPos)
            return self.IncrementalWalk(direction, objects, distance-1)
        else:
            for i in objects:
                if self != i:
                    if self.DoesOverlap(i):
                        self.SetPos(tempPos)
                        return self.IncrementalWalk(direction, objects, distance-1)
            return True

    # Draws a sprite that damages any enemy it touches if the character can attack
    def Attack(self, objects, display, AI):
        if self.attack.hold:
if self.CanAttack():
    if not self.attack.hold:
        for i in objects:
            if i.flammable:
                if self.attack.DoesOverlap(i) and i != self:
                    i.AdjustAttribute("Health", -1)
                    if i.enemy:
                        AI.currentSuccessValue += -1
                    else:
                        AI.currentSuccessValue += 2
            if i.GetHealth() == 0:
                objects.remove(i)
        return(("Something", "burned.",""))
    return False
else:
    return False

#The class the holds each enemy character's stats and actions
class Enemy(Character):
    def __init__(self, type):
        self.type = type
        if self.type == "spider":
            img1 = "/Art/Spider.png"
            img2 = "/Art/Spider.png"
            Character.__init__(self,img1,img2)
            self.attributes = [10,10,10,18]
        elif self.type == "lizard":
            img1 = "../Art/AquaLizard.png"
            img2 = "../Art/AquaLizard.png"
            Character.__init__(self,img1,img2,(150,150))
            self.attributes = [13,10,13,13]
        else:
            img1 = "../Art/Rock1.png"
            img2 = "../Art/Rock1.png"
            Character.__init__(self,img1,img2)
            self.attributes = [20,5,15,5]
            self.attack = Attack("../Art/Slash.png", False, (50,50))
            self.move = 0
            self.enemy = True
            self.wander = 0
            self.status = 0
            self.action = ""
            self.lastAction = ""
            self.previousHealth = self.attributes[0]
            self.maxHealth = self.attributes[0]

    #Whether this is using the neural network AI or using the basic AI
    self.usingAI = True

    def Act(self, player, OBJECTS, AI, display):
        #choose action and perform it
        #Actions: attack, flee, turn towards and approach the player, and wander
self.lastAction = self.action

self.action = self.ChooseAction(player, OBJECTS, AI)

if self.action == "combat":
    self.Combat(player, OBJECTS, AI, display)
elif self.action == "flee",
    self.Flee(player, OBJECTS)
elif self.action == "turn",
    self.Turn(player, OBJECTS)
else:
    self.Wander(OBJECTS)

def ChooseAction(self, player, OBJECTS, AI):
    # returns action to do
    if self.usingAI:
        if self.status == 0:
            move = AI.extrapolate(AI.newDataPoint(self, player, OBJECTS))
            if move[0] > .5:
                self.status = 5
                return "combat"
            elif move[1] > .5:
                self.status = 10
                return "turn"
            elif move[2] > .5:
                self.status = 5
                return "flee"
            else:
                self.status = 10
                return ""
        else:
            self.status = self.status - 1
            return self.action
    else:
        distance = self.DistanceTo(player)
        if self.type == "rock"
            if self.status == 0:
                if distance >= 50 or
                    self.GetDirection() != self.DirectionTo(player):
                    self.status = 10
                    return "turn"
                else:
                    self.status = 10
                    return "combat"
            else:
                self.status += -1
                return self.action
        elif self.status == 0:
            if distance <= 500:
                if self.previousHealth > self.GetHealth():
                    self.status = 5
                    self.previousHealth = self.GetHealth()
                    return "flee"
                elif distance >= 50 or
                    self.GetDirection() != self.DirectionTo(player):
                    self.status = 5
                    return "turn"
                else:
                    self.status = 10
                    return "combat"
else:
    self.status = 10
    return ""
else:
    self.status += -1
    return self.action

def Combat(self, player, OBJECTS, AI, display):
    # attack player
    self.Attack(OBJECTS, display, AI)

def Flee(self, player, OBJECTS):
    # move away from player
    self.Walk(self.DirectionTo(player)+4, OBJECTS)

def Turn(self, player, OBJECTS):
    # changes angle and walks towards player
    if self.status == 0:
        self.SetDirection(self.DirectionTo(player))
        for i in range(7):
            if not self.Walk(self.GetDirection(), OBJECTS) and not self.CanInteract(player):
                if i%2 == 1:
                    self.Walk(self.GetDirection()-i, OBJECTS)
                else:
                    self.Walk(self.GetDirection()+i, OBJECTS)
        else:
            break

def Wander(self, OBJECTS):
    # move around randomly
    self.move+=1
    if self.move==10:
        self.move=0
    for i in OBJECTS:
        if self.CanInteract(i):
            self.move = 0
            break
    if self.move==0 or self.move==5:
        self.wander = numpy.random.randint(-3,3)
        if not self.Walk(self.wander, OBJECTS):
            self.move = -1

# Makes decisions for the enemy characters

class NeuralNetworkAI():
    def __init__(self, initialInputData, initialOutputData, learningRate = .1):
        self.totalData = len(initialInputData)
        self.dataLength = len(initialInputData[0]) + 1
        numpy.random.seed(1)
        self.weights1 = 2*numpy.random.random((self.dataLength,self.totalData)) - 1
        self.weights2 = 2*numpy.random.random((self.totalData,self.totalData)) - 1
        self.weights3 = 2*numpy.random.random((self.totalData,4)) - 1
        self.inputData = initialInputData
        # takes an array of input data, changed to values between 0 and 1

#each array is length 3 in the form of:
# [current_direction, player_distance, player_direction]

# How to calculate each input:
current_direction = (self.GetDirection() + 4) / 8
player_distance = self.DistanceTo(player) / 1500
player_direction = (self(DirectionTo(player) + 4) / 8

self.outputData = initialOutputData

# takes an array of output data for each input point, of values 0 to 1
self.unclaimedData = numpy.array([[5]])

# will be used to store collected data with undetermined success value
self.unclaimedResults = numpy.array([[5]])

self.currentSuccessValue = 0

self.learningRate = learningRate

def trainNeuralNetwork(self, times):
    10 = []
    for i in range(self.totalData):
        if i == 0:
            10 = numpy.append(self.inputData[0], [1])
        else:
            10 = numpy.append(10, [numpy.append(self.inputData[i], [1], 0)], 0)

    for i in xrange(times):
        11 = self.sigmoid(numpy.dot(10, self.weights1))
        12 = self.sigmoid(numpy.dot(11, self.weights2))
        13 = self.sigmoid(numpy.dot(12, self.weights3))

        13_error = self.outputData - 13
        13_delta = 13_error * self.sigmoid(13, deriv=True)

        12_error = 13_delta.dot(self.weights3.T)
        12_delta = 12_error * self.sigmoid(12, deriv=True)

        11_error = 12_delta.dot(self.weights2.T)
        11_delta = 11_error * self.sigmoid(11, deriv=True)

        self.weights3 += 12.T.dot(self.learningRate * 13_delta)
        self.weights2 += 11.T.dot(self.learningRate * 12_delta)
        self.weights1 += 10.T.dot(self.learningRate * 11_delta)

    def newDataPoint(self, char, player, OBJECTS):
        current = numpy.array([(char.GetDirection() + 4) / 8])
        current = numpy.append(current, [(char.DistanceTo(player)) / 1500, (char.DirectionTo(player) + 4) / 8])

        return current

    def collectData(self, character, player, OBJECTS, Refresh=False):
        if self.unclaimedData[0][0] == 5:
            self.unclaimedData = numpy.array([self.newDataPoint(character, player, OBJECTS)])
else:
    self.unclaimedData = numpy.append(self.unclaimedData, 
        [self.newDataPoint(character, player, OBJECTS)], axis = 0)

if character.action == "combat":
    act = [1,0,0,0]
elif character.action == "turn":
    act = [0,1,0,0]
elif character.action == "flee":
    act = [0,0,1,0]
else:
    act = [0,0,0,1]

if self.unclaimedResults[0][0] == 5:
    self.unclaimedResults = numpy.array([act])
else:
    self.unclaimedResults = numpy.append(self.unclaimedResults, [act], 
        axis = 0)

if Refresh:
    if self.currentSuccessValue < 0:
        self.unclaimedData = numpy.array([5])
        self.currentSuccessValue = 0
        self.unclaimedResults = numpy.array([5])
    else:
        tempInputs = numpy.append(self.unclaimedData, self.inputData, 
            axis = 0)
        tempOutputs = numpy.append(self.unclaimedResults, 
            self.outputData, axis = 0)
        tempOrg = numpy.random.choice(len(tempInputs), size = 
            self.totalData, replace = False)
        self.inputData = numpy.array([5])
        for i in range(self.totalData):
            if self.inputData[0][0] == 5:
                self.inputData = numpy.array([tempInputs[tempOrg[i]]])
                self.outputData = 
                    numpy.array([tempOutputs[tempOrg[i]]])
            else:
                self.inputData = numpy.append(self.inputData, 
                    [tempInputs[tempOrg[i]]], axis = 0)
                self.outputData = numpy.append(self.outputData, 
                    [tempOutputs[tempOrg[i]]], axis = 0)
        self.unclaimedData = numpy.array([5])
        self.unclaimedResults = numpy.array([5])
        self.trainNeuralNetwork(50)

# forward propagation
    def extrapolate(self,newInput):
        l0 = numpy.append(newInput, 1)
        l1 = self.sigmoid(numpy.dot(l0, self.weights1))
        l2 = self.sigmoid(numpy.dot(l1, self.weights2))
        l3 = self.sigmoid(numpy.dot(l2, self.weights3))
        return l3
# sigmoid function - converts the outputs from each layer non-linearly into a number from 0-1
# if using the derivative, a high derivative indicates more uncertainty

def sigmoid(self, x, deriv=False):
    if deriv:
        return x*(1-x)
    return 1/(1+numpy.exp(-x))

# Inanimate Objects Class - class for things that won't move on their own - rocks and trees and such.
# Might be able to break them.
class Inanimate(Sprite):
    def __init__(self, img, sturdy, size = (150,150)):
        Sprite.__init__(self, img, size)
        self.sturdy = sturdy
        self.broken = False

# Attack Class - Used to display attacks
class Attack(Sprite):
    def __init__(self, img, hold, size = (50,50)):
        Sprite.__init__(self, img, size)
        self.substantial = False
        self.hold = hold
        self.image1 = self.image
        self.image2 = pygame.transform.scale(self.image1, (self.size[0], self.size[1]))
        self.image2 = pygame.transform.flip(self.image2, True, False)
        self.state = 0

    def Show(self, source, sourceSize, direction, display):
        self.SetDirection(direction)
        delta = self.Delta(direction, (sourceSize[0]+self.size[0])/2, (sourceSize[1]+self.size[1])/2)
        self.Place((source[0]+delta[0]+(sourceSize[0]-self.size[0])/2,source[1]+delta[1]+(sourceSize[1]-self.size[1])/2),display)
        self.state += 1
        if self.state >= 4:
            self.image = self.image1
            self.state = 0
        elif self.state >= 2:
            self.image = self.image2

# Door Class - Interacting with one passes you to another room, unless a key is required.
class Door(Sprite):
    def __init__(self, img, nextRoom, opensTo, size = (50,150), locked = False):
        Sprite.__init__(self, img, size)
        self.nextRoom = nextRoom
        self.interactable = True
        self.locked = locked
        self.opensTo = opensTo
        if self.locked:
            self.text = ("Door is", "locked.","")
        else:
            self.text = ("Opening", "door...",")

    def Place(self, source, sourceSize, direction, display):
        self.SetDirection(direction)
        delta = self.Delta(direction, (sourceSize[0]+self.size[0])/2, (sourceSize[1]+self.size[1])/2)
        self.Place((source[0]+delta[0]+(sourceSize[0]-self.size[0])/2,source[1]+delta[1]+(sourceSize[1]-self.size[1])/2),display)
        self.state += 1
        if self.state >= 4:
            self.image = self.image1
            self.state = 0
        elif self.state >= 2:
            self.image = self.image2
def IsLocked(self):
    return self.locked

def Unlock(self):
    self.locked = False

def Open(self):
    return self.nextRoom, self.opensTo

#Words class - Used to display text
class Words(Sprite):
    def __init__(self, text, position, size):
        Sprite.__init__(self, '..', '/Art/Words.png', (size, size))
        self.size = size
        self.position = position
        self.substantial = False
        self.text = text  # pass a tuple as text to look through it for each line
        self.font = pygame.font.SysFont('bradleyhandic', size, True)

    def Place(self, position, surface, color=(0, 0, 0)):
        placement = 0
        self.SetPos(position)
        for i in self.text:
            words = self.font.render(i, True, color)
            surface.blit(words, (position[0], position[1] + placement))
            placement += self.size + 10

    def SetText(self, text):
        text = list(text)
        text.reverse()
        for i in text:
            self.text = [i] + self.text
        if len(self.text) > 11:
            self.text.pop()

    def __init__(self, text, position, size):
        Sprite.__init__(self, '..', '/Art/Words.png', (size, size))
        self.size = size
        self.position = position
        self.substantial = False
        self.text = text  # pass a tuple as text to look through it for each line
        self.font = pygame.font.SysFont('bradleyhandic', size, True)

    def Place(self, position, surface, color=(0, 0, 0)):
        placement = 0
        self.SetPos(position)
        for i in self.text:
            words = self.font.render(i, True, color)
            surface.blit(words, (position[0], position[1] + placement))
            placement += self.size + 10

    def SetText(self, text):
        text = list(text)
        text.reverse()
        for i in text:
            self.text = [i] + self.text
        if len(self.text) > 11:
            self.text.pop()
rock1.Place((300,300), DISPLAYSURF)
rock2 = Inanimate('../Art/Rock1.png', 10, (150,150))
rock2.SetInteractable(True)
rock2.SetText(("It's a ", "large rock",""))
rock2.Place((400,400), DISPLAYSURF)

rock3 = Inanimate('../Art/Rock1.png', 10, (50,50))
rock3.SetInteractable(True)
rock3.SetText(("It's a", "small rock",""))
rock3.SetDirection(-3)
rock3.Place((1100,150), DISPLAYSURF)

wall1 = Inanimate('../Art/Wall1.png', 100, (1250,50))
wall1.Place((0,0), DISPLAYSURF)

wall2 = Inanimate('../Art/Wall1.png', 100, (1250,50))
wall2.Place((0, DISPLAY_Y - 50), DISPLAYSURF)

wall3 = Inanimate('../Art/Wall2.png', 100, (50,1000))
wall3.Place((0, 0), DISPLAYSURF)

wall4 = Inanimate('../Art/Wall2.png', 100, (50,1000))
wall4.Place((DISPLAY_X - 300, 0), DISPLAYSURF)

wall5 = Inanimate('../Art/Wall2.png', 100, (50, 700))
wall5.Place((900,0), DISPLAYSURF)

spider = Enemy("spider")
spider.SetInteractable(True)
spider.SetText(("AAAAAA","AAAAAA","SPIDER",""))
spider.Place((700,500), DISPLAYSURF)

spider2 = Enemy("spider")
spider2.SetInteractable(True)
spider2.SetText(("AAAAAA","it's another","spider",""))
spider2.Place((500,700), DISPLAYSURF)

spider3 = Enemy("spider")
spider3.SetInteractable(True)
spider3.SetText(("AAAAAA","it's another","spider",""))
spider3.Place((400,800), DISPLAYSURF)

morph = Enemy("rock")
morph.SetInteractable(True)
morph.SetText(("It's a rock", "... or", "is it?",""))
morph.Place((700,500), DISPLAYSURF)

lizard = Enemy("lizard")
lizard.SetText(("Uh... It's", "a lizard...","... maybe...",""))
lizard.Place((600,500), DISPLAYSURF)

intro = Words(['Use W A S D keys to move', 'Press LEFT SHIFT to interact', 'Press SPACE to attack'], (250,100), 40)

room1 = [intro, morph, rock1, rock3, wall1, wall2, wall3, wall4]
room2 = [spider, rock1, rock2, wall1, wall2, wall3, wall4, wall5]
room3 = [lizard, rock2, rock3, wall1, wall2, wall3, wall4]
room4 = [spider2, spider3, rock3, wall1, wall2, wall3, wall4]
door1 = Door('../Art/DoorVertical.png', room2, (50, DISPLAY_Y/3))
door1.Place((DISPLAY_X - 300, DISPLAY_Y/3), DISPLAYSURF)
room1.append(door1)
room1.append(player)
door2 = Door('../Art/DoorVertical.png', room1, (DISPLAY_X - 400, DISPLAY_Y/3))
door2.Place((0, DISPLAY_Y/3), DISPLAYSURF)
room2.append(door2)
room2.append(player)
door3 = Door('../Art/DoorVertical.png', room3, (50, 2*DISPLAY_Y/3))
door3.Place((DISPLAY_X - 300, 2*DISPLAY_Y/3), DISPLAYSURF)
room2.append(door3)
door4 = Door('../Art/DoorVertical.png', room2, (DISPLAY_X - 400, 2*DISPLAY_Y/3))
door4.Place((0, 2*DISPLAY_Y/3), DISPLAYSURF)
room3.append(door4)
door5 = Door('../Art/DoorHorizontal.png', room4, (DISPLAY_X/3, 50), (150,50))
door5.Place((DISPLAY_X/3, DISPLAY_Y-50), DISPLAYSURF)
room3.append(door5)
room3.append(player)
door6 = Door('../Art/DoorHorizontal.png', room3, (DISPLAY_X/3, DISPLAY_Y-150), (150,50))
door6.Place((DISPLAY_X/3, 0), DISPLAYSURF)
room4.append(door6)
room4.append(player)

OBJECTS = room1

enemies = [morph, spider, lizard, spider2, spider3]
for i in enemies:
    i.AdjustAttribute("Attack",1)

player.level = passedLevel

health = Words([str(player.GetHealth())], (1325, 95), 40)
level = Words([str(player.level)], (1325, 270), 40)
enemies_killed_str = Words([str(enemies_killed)], (1325, 425), 40)
text = Words([""], (1310, 560), 25)

STATS = (health, level, enemies_killed_str, text)

DISPLAYSURF.fill(BGCOLOR)

dead = False
justdied = True

while True: # game loop
    if dead:
        DEATH_SCREEN = Words(['   You have died',"Click anywhere to play again."], (250,100), 50)
if justdied:
    DISPLAYSURF.fill(BGCOLOR, None, BLEND_RGBA_ADD)
    justdied = False
DEATH_SCREEN.Place(DEATH_SCREEN.GetPos(), DISPLAYSURF)
for event in pygame.event.get():
    if event.type == QUIT:
        pygame.quit()
        sys.exit()
    elif event.type == MOUSEBUTTONUP:
        main(AI, refreshCount, player.level, enemies_killed)
    else:
        mouseClicked = False
        INTERACT = False
        DISPLAYSURF.blit(bg, (0,0))
for event in pygame.event.get():
    if event.type == QUIT:
        pygame.quit()
        sys.exit()
    elif event.type == MOUSEBUTTONUP:
        mouseClicked = True
    elif event.type == KEYDOWN:
        if event.key == K_LSHIFT:
            INTERACT = True
    if mouseClicked:
        print(pygame.mouse.get_pos())
    keys = pygame.key.get_pressed()
    LEFT = keys[K_a]
    RIGHT = keys[K_d]
    UP = keys[K_w]
    DOWN = keys[K_s]
    ATTACK = keys[K_SPACE]
    if UP:
        if LEFT:
            player.Walk(-3, OBJECTS)
        elif RIGHT:
            player.Walk(-1, OBJECTS)
        else:
            player.Walk(-2, OBJECTS)
    elif DOWN:
        if LEFT:
            player.Walk(3, OBJECTS)
        elif RIGHT:
            player.Walk(1, OBJECTS)
        else:
            player.Walk(2, OBJECTS)
    elif LEFT:
        player.Walk(4, OBJECTS)
    elif RIGHT:
        player.Walk(0, OBJECTS)
    if INTERACT:
        for i in OBJECTS:
if i.interactable:
    if player.CanInteract(i):
        text.SetText(i.GetText())
        if isinstance(i, Door):
            if not(i.IsLocked()):
                OBJECTS = i.Open()[0]
                player.SetPos(i.Open()[1])
                break
        for i in OBJECTS:
            if i.enemy == True:
                i.Act(player, OBJECTS, AI, DISPLAYSURF)
                i.IncrementAttack()
                refreshCount += 1
                if refreshCount >= 60:
                    AI.collectData(i, player, OBJECTS, True)
                    refreshCount = 0
                elif refreshCount%10 == 0:
                    AI.collectData(i, player, OBJECTS)
                    i.Place(i.GetPos(), DISPLAYSURF)
            if ATTACK:
                attack = player.Attack(OBJECTS, DISPLAYSURF, AI)
                if attack != False:
                    text.SetText(attack)
                    enemies_killed += 1
                    enemies_killed_str.text= [str(enemies_killed)]
                    if enemies_killed % 5 == 0:
                        player.level += 1
                        level.text = [str(player.level)]
                        for i in enemies:
                            i.AdjustAttribute("Attack",1)
                            for i in enemies:
                                if i.GetHealth() > 0:
                                    break
                                elif i == enemies[-1]:
                                    print i
                                    for j in enemies:
                                        j.AdjustAttribute("Health", j.maxHealth)
                                        room1.append(morph)
                                        room2.append(spider)
                                        room3.append(lizard)
                                        room4.append(spider2)
                                        room4.append(spider3)
                                        text.SetText(('Enemies have respawned.',''))
                                        if player.GetHealth()<=0:
                                            text.SetText(('You died.',''))
                                            dead = True
                                            health.text = ((str(player.GetHealth()),''))
                                            for i in STATS:
                                                i.Place(i.GetPos(),DISPLAYSURF)
                                            player.IncrementAttack()
                                            pygame.display.update()
                                            fpsClock.tick(FPS)
#Initializing A.I. with tested parameters

```python
inputArray = numpy.random.random((100,3))
outputArray = numpy.random.choice(numpy.array([0,1]),(100,4))

inputArray = numpy.array([[0.25, 0.155688448415, 0.75],
[0.0, 0.359467375886, 1.0],
[0.875, 0.205917339058, 0.875],
[0.375, 0.0244215663285, 0.375],
[0.25, 0.122981080927, 0.75],
[1.0, 0.0445640717722, 0.5],
[0.75, 0.00174918873308, 0.75],
[0.125, 0.40978027009, 0.125],
[0.25, 0.00162555084805, 0.25],
[0.125, 0.31369290117, 0.125],
[0.75, 0.0169856590862, 0.75],
[0.125, 0.250979347249, 0.125],
[1.0, 0.069356639673, 0.625],
[1.0, 0.0183547828482, 1.0],
[0.0, 0.442144584082, 0.125],
[1.0, 0.0283425991062, 1.0],
[0.0, 0.137838199269, 0.25],
[0.625, 0.0859784836301, 0.625],
[0.0, 0.442144584082, 0.125],
[0.125, 0.148542886138, 0.125],
[0.625, 0.023392154901, 0.625],
[0.0, 0.248126210978, 0.25],
[0.75, 0.0842253858378, 0.5],
[1.0, 0.18073343682, 1.0],
[0.0, 0.360933873884, 0.125],
[0.875, 0.010614069676, 0.875],
[0.0, 0.061383174512, 0.25],
[0.875, 0.0647056672056, 0.75],
[0.625, 0.093924493284, 0.625],
[0.5, 0.0185975645382, 0.75],
[0.25, 0.0692665914232, 0.125],
[0.125, 0.435661582631, 0.125],
[0.5, 0.0044696169736, 0.5],
[0.5, 0.0044696169736, 0.5],
[0.125, 0.180053798161, 0.125],
[0.5, 0.136607806483, 0.625],
[0.125, 0.377559289827, 0.125],
[0.25, 0.255604542668, 0.75],
[0.375, 0.00162766355362, 0.5],
[0.125, 0.34551528608, 0.125],
[0.625, 0.10962788442, 0.5],
[0.75, 0.106793408048, 0.75],
[0.0, 0.377233098106, 0.25],
[0.75, 0.0449779700156, 0.625],
[0.75, 0.112535012735, 0.75],
[0.0, 0.167801257161, 0.125],
[0.0, 0.314401177048, 0.875],
[0.375, 0.0942999943698, 0.875],
[0.75, 0.000450780762508, 0.75],
[0.625, 0.0349267322222, 0.625],
[0.75, 0.0828922756579, 0.625],
```
```
0], [1, 0, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [1, 0, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0])

977.
978.AI = NeuralNetworkAI(inputArray, outputArray, .05)
979.AI.trainNeuralNetwork(10000)
980.refreshCount = 0
981.
982.main(AI, refreshCount)

7.2 Test Functions

Test Networks 1 and 2 - used for comparing the efficacy of different numbers of hidden layers

1. def testNetwork1(self):
2.(numpy.random.seed(1)
3. testInitial = numpy.array([[0, .5, 1, 0],
4. [0, 0, 1, 0],
5. [1, 0, 1, 0],
6. [1, 1, .5, 0]])
7. testResult = numpy.array([[.75],
8. [1],
9. [1],
10. [.83]])
11. randomWeights1 = 2*numpy.random.random((4,4)) - 1
12. randomWeights2 = 2*numpy.random.random((4,1)) - 1
13. for i in xrange(10000):
14. 10 = testInitial
15. 11 = self.sigmoid(numpy.dot(10,randomWeights1))
16. 12 = self.sigmoid(numpy.dot(11,randomWeights2))
17. 12_error = testResult - 12
18. if (i % 1000) == 0:
19.  print "Error:" + str(numpy.mean(numpy.abs(12_error)))
20. 12_delta = 12_error*self.sigmoid(12,deriv=True)
21. 11_error = 12_delta.dot(randomWeights2.T)
22. 11_delta = 11_error*self.sigmoid(11,deriv=True)
23. randomWeights2 += 11.T.dot(12_delta)
24. randomWeights1 += 10.T.dot(11_delta)
36.    print l2
37. 38. def testNetwork2(self):
39.    numpy.random.seed(1)
40. 41. testInitial = numpy.array([[0,.5,1,0],
42.        [0,0,1,0],
43.        [1,0,1,0],
44.        [1,1,.5,0]])
45. 46. testResult = numpy.array([[.75],
47.        [1],
48.        [1],
49.        [.83]])
50. 51. randomWeights1 = 2*numpy.random.random((4,4)) - 1
52. randomWeights2 = 2*numpy.random.random((4,4)) - 1
53. randomWeights3 = 2*numpy.random.random((4,1)) - 1
54. 55. for i in xrange(10000):
56.    10 = testInitial
57.    11 = self.sigmoid(numpy.dot(l0,randomWeights1))
58.    12 = self.sigmoid(numpy.dot(l1,randomWeights2))
59.    13 = self.sigmoid(numpy.dot(l2,randomWeights3))
60.    13_error = testResult - l3
61. 62.    if (i% 1000) == 0:
63.       print "Error:" + str(numpy.mean(numpy.abs(l3_error)))
64. 65.    13_delta = 13_error*self.sigmoid(l3,deriv=True)
66. 67.    12_error = 13_delta.dot(randomWeights3.T)
68. 69.    12_delta = 12_error*self.sigmoid(l2,deriv=True)
70. 71.    11_error = 12_delta.dot(randomWeights2.T)
72. 73.    11_delta = 11_error*self.sigmoid(l1,deriv=True)
74. 75.    randomWeights3 += 12.T.dot(13_delta)
76. randomWeights2 += 11.T.dot(12_delta)
77. randomWeights1 += 10.T.dot(11_delta)
78. 79. print l3

Test Neural Network - used to compare different learning rates and network sizes

1. def testNN(inputArray,outputArray):
2.    testing = NeuralNetworkAI(inputArray,outputArray,.05)
3. 4. testing.trainNeuralNetwork(5000)
5. 6. print "Extrapolated: " + str(testing.extrapolate(inputArray[0]))
7. print "Expected: " + str(outputArray[0])
8. print "Difference: " + str(outputArray[0] - testing.extrapolate(inputArray[0])) + "\n"
9. 
10. print "Extrapolated: " + str(testing.extrapolate(inputArray[1]))
11. print "Expected: " + str(outputArray[1])
12. print "Difference: " + str(outputArray[1] - testing.extrapolate(inputArray[1])) + "\n"
13. 
14. print "Extrapolated: " + str(testing.extrapolate(inputArray[6]))
15. print "Expected: " + str(outputArray[6])
17. 
18. print "Extrapolated: " + str(testing.extrapolate(inputArray[8]))
19. print "Expected: " + str(outputArray[8])
20. print "Difference: " + str(outputArray[8] - testing.extrapolate(inputArray[8])) + "\n"
21. 
22. print "Extrapolated: " + str(testing.extrapolate(inputArray[93]))
23. print "Expected: " + str(outputArray[6])
24. print "Difference: " + str(outputArray[6] - testing.extrapolate(inputArray[6])) + "\n"
25. 
26. print "Extrapolated: " + str(testing.extrapolate(inputArray[95]))
27. print "Expected: " + str(outputArray[8])
28. print "Difference: " + str(outputArray[8] - testing.extrapolate(inputArray[8])) + "\n"
7.3 Survey

1. How did you find fighting the enemies in the first version?
   Too hard    Somewhat hard    Just right    Somewhat easy    Too easy

2. In the second version?
   Too hard    Somewhat hard    Just right    Somewhat easy    Too easy

3. How long did you play the game for? ________________

4. How would you score the game play?
   Poor 1 2 3 4 5 Excellent

5. How would you score the graphics design?
   Poor 1 2 3 4 5 Excellent

6. Do you have any suggestions for future progression of this game or any other comments?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
Bibliography


