Vector Supervisor Engine System

Bradley Orego
Aaron Gorenstein
Andrew Wood
Jake Scheiber
Greg Wilbur
Robert Yoon
Sara Melnick
Cole Brown
Frank Ferraro
Evan VanDeGriff

University of Rochester
Department of Computer Science

A SlytherClaw Product

March 25, 2009

Abstract

The game of Vector is a board game played by 2-4 players in which a single playing piece is manipulated about a board using moves determined by vector addition of each player. A computerized version of this game can be built such that 4 computer players can view the game state, understand the current status, and make intelligent calculations and moves which attempt to maximize their score, winning them the game, and subsequently the tournament. An integrated GUI system shows bystanders both a live-action show of the moves, as well as provides a turn-by-turn replayable history of each game. Tight network protocols provide for flawless communication between server and client, allowing for an enjoyable experience by both players and onlookers.

1 Intro

Game engines are some of the most complicated pieces of software in development today. Game development is an increasingly popular and complex field of software, with new technologies and practices being created every day. Most of the work on game engines today focuses on newer and bigger games, leaving the simple pleasures of board games by the wayside. One of these such games, Vector, has almost fallen completely off the map in terms of board game entertainment. Our project hopes to bring this future classic back to popularity.

The VSE (Vector Supervisor Engine) is a standalone application designed to work with 4 clients (human or computer) via a TCP/IP network connection. The VSE is coded to uphold all of the rules of the Vector board game, as well as several different options relating to the communications. VSE is designed to run a tournament of 13 games, where each game has a maximum of 12 turns. Each tournament has a randomized placement of the four clients into the four Vector playing positions per game. Play continues as it would in a real-life game of Vector, using the network protocol (see Appendix A for details) as opposed to physical playing cards.

2 Protocol

The network protocol was designed to be as unambiguous and explicit as possible. The VSE begins the tournament by querying for a client identifier, waiting until 4 clients have connected. After all players have connected, the VSE randomizes game order (that is, for each game the VSE randomly determines which board position each player will be playing for that game), and prints this information to each client, so one can identify where they are positioned for each game. This begins the game loop. The game loop essentially runs until the end of the game. At the start of the game loop, the number of the game is identified, and reminds each client which position they are playing. Then, the round loop starts, which is what actually drives the game. At the start of each round (or "turn"), the turn number is identified, as well as the position of the Vector playing piece, and the play order for that turn. From there, the VSE queries each player for a direction (in sequence), reports each direction to all players before querying the next. Then, the VSE opens simultaneous queries to all players for magnitudes, reporting the magnitudes only after all magnitudes have been received. After receiving and displaying all of this information, the VSE executes the turn, updates scores, and begins the next turn. If there are ever disqualifications, the loop will be interrupted, and the disqualification will be announced, ending the game and starting the next game. At the end of the game, tournament scores will be updated and displayed, and at the end of the tournament, a winner will be announced. Please see Appendix D for a complete

protocol.

3 Design

Programming the VSE required a slew of design decisions, both in terms of design and implementation. The design team was divided into four main parts: Engine, Networking, Graphics/UI, and Testing. Figure 1 briefly describes the design breakup. The VSEs main component is the Tournament Supervisor, which is split between a Game Logic module and a Server module, who talk to each other within the Supervisor. Each of the four clients communicate with the Supervisor via TCP/IP, and the Supervisor also passes information to the GUI, which holds the board state, and also passes this information to the sidebar which updates scores and other game information. The basic overview

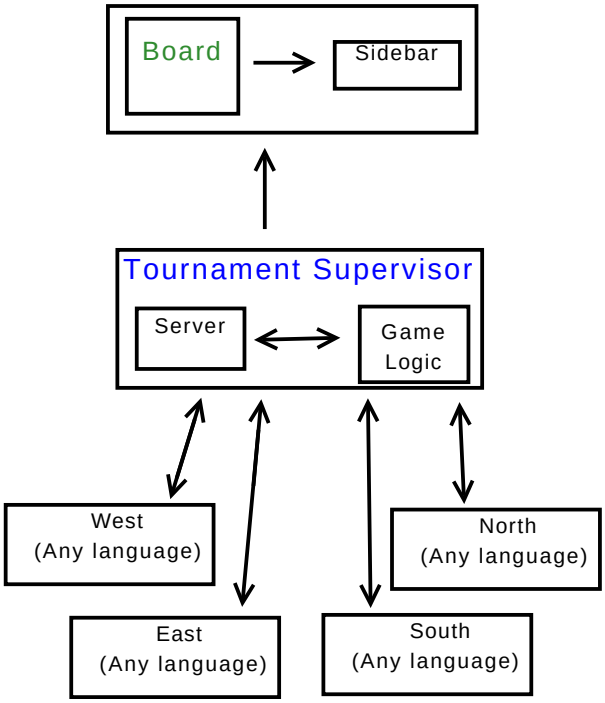

Figure 1: A diagram of how VSE is designed.

of each division is as follows: the Engine team was in charge of designing and programming the actual engine that runs the game. The Engine contains all of the game logic, as well as handles all of the regulations, such as malformed input and time constraints. The Networking team was in charge of handling

the socketing and networking issues, as well as the development of the protocol. Graphics/UI handled anything relating to the visual representation of the game state, and the Testing group responsible for creating dummy test cases to checks for weaknesses and tests robustness of the VSE system. Each individual group had specific design decisions to make, but there are also over-arching decisions made for the entire project.

The system is all developed in Python for several reasons. Python has excellent support for networking, has relatively low overhead in terms of programming and development, and easy object-orientation. Python has a package, known as PyGame, which also has very easy and effective graphics support, and flawless integration with Python. Also, in terms of compilation and exportation, there is a easy-to-compile, easy-to-run executable using a compilation program known as `cxFreeze`, which allows for Python files to be "frozen" as an executable, so that it can run on all systems (notably, those without Python and/or PyGame installed). All of these factors lead to the decision to stick with Python as our main programming language, and each group decided to stick to the standard in order to make interfacing between the groups simple and straightforward. The system design was facilitated by [1], which was a very excellent resource for all things relating to Python.

4 Committee Projects

Trying to tackle the entire project as one chunk would be both impractical and inefficient. The best way to address the various concerns of this project was to identify the main sub-projects, and to delegate people to work on each project. Given the specifications of the project, and the requirements of the entire system, the division was somewhat organic. Each of the groups was then capable to work mostly independently of each other, allowing for human-level parallel processing. Intercommunication amongst the groups was facilitated by one main Project Manager, which organized meetings and ensured clear communication amongst groups.

4.1 Engine/Game Logic

VSEs Engine team discovered various difficulties in their development, and came up with different qualities and features of the system. One of the first problems the engine team ran into was how to create an internal representation of the board. The solution for this was to have a wrapper class read in an ASCII representation of the board, parse that input and save the board as a two-dimensional array of board squares. The ASCII representation is fairly intuitive; reading the board in row-major order, simply step across the columns and write

(in English) any information that particular square may contain. For instance, if the square is a yellow move square, the representation is just `MOVE NORTH 3`. Each square is limited to a new line (i.e. a carriage return), and each row must contain 21 squares. This representation made it very easy to implement the logic later on, and it was also very user-friendly. As is convention with two-dimensional arrays, the piece location is referenced by `row` and then `column`.

The other difficulties the engine team ran into involved mostly game logic issues and following the rules laid out in [2]. Being able to determine when a goal was scored was the first main concern encountered. Initially, it was possible to move into and out of a goal in the process of a turn without registering the goal state. The solution of this is to "keep track" of the game state after each physical move. The secondary concern came in determining when a piece fell off the board. The initial problem was that a piece would move off the board by more than one square, would be relocated to the respective corner, and would continue its movement, instead of stopping and losing the next turn. By normalizing the movement to be at most 1 square off the board, it solves this problem. Determining the loss of turns and disqualifications was also an Engine problem. The issue here involved cascading effects (that is, more than one person can lose a turn/be disqualified at the same time), and this was solved by creating multiple arrays and cross-referencing between rounds, updating each array as necessary. This is not the most efficient or elegant solution, but it accomplishes the goal.

4.2 Networking

The Networking team had several hiccups in their development as well. The formal protocol was one of the most important tasks of this team, and having an unambiguous and explicit network protocol is essential to the application and execution of this project. The protocol was designed to be intuitive and user-friendly. Aside from that, dealing with latencies and other communication issues within the actual program were concerns of the Network team. Given the scope of this project, network latency is not a main issue (as the entire tournament will be on a local network), and the timing can be reliably measured by Pythons internal `time.time()` function. Use of non-blocking I/O, which is very easy to implement in Python, allowed for simultaneous checking of players exceeding the time limit, which also facilitated simultaneous disqualifications as necessary.

4.3 Graphics/UI

The Graphics/UI team used pre-rendered images for the board and cards, utilizing PyGames packages, and PyGames text objects to render labels and scores; examples of some of the pre-rendered images can be found in Appendix C.

The UI window is split into a "board" area and a "sidebar" area, where the game state is portrayed using the images on the board, and the game and tournament scores are displayed on the sidebar. UI communication comes via messages passed to the UI controller from the Engine. Messages are passed by two lists a primary list of direction and magnitude cards, and a secondary list with the outcomes of the turn in regards to board state, movements made, and scores. Vector piece movement is portrayed by placement of colored arrows on the board, corresponding to each individual player. North will be codified as Red, South as Blue, East as Yellow, West as Green. Any movements caused by the board will be portrayed in the color of the player that caused the vector to end on that movement square. The UI has two modes of display real-time and slideshow. Real-time mode displays the moves as they occur on the board, updating as the game plays. At the end of the game, the UI enters "slideshow" mode. In the slide show, there are two options for viewing. Pressing the up or down key will skip individual games, showing only the end result of each game. Left and right arrows cycle through each individual turn, showing the game as it progressed. Each turn can be reconstructed using the arrows, which are redrawn at each step in the "slideshow." Pressing the escape key will quit slideshow mode, and proceed with the next game in the tournament. The only main problem with the GUI is there is an issue rendering the first gameplay arrow of the first turn of any game, making 13 missing arrows in the course of a tournament. The issue is that the game thinks the starting point is wherever the piece was previously (at the end of the last game - i.e. in the goal), which causes it to draw the arrow offset by however far the endpoint was from the center.

4.4 Testing

The Testing team was charged with the task of creating many different agents to test for weaknesses of the Engine section. The idea was for test cases to emit random behavior (as we do not actually care what the outcome of the game is), but providing erroneous input or breaking the time limit condition with some randomized fixed probability. The hopes were to create test agents in as many different languages that we could, so that we could provide a prototype for future designers. There are completed (random) agents for Python, C, and Java, providing an effective test case for the VSE. These agents accomplished their task, as they pointed out flaws in various parts of the system.

5 Conclusion and Future Work

VSE is an effective and working computerized version of the board game, Vector. There are many design decisions that were made to ensure functionality as op-

posed to efficiency, as well as functionality over elegance of code/design. Future work for our project would involve making a more robust, efficient server. It is assumed that there are no intents to cheat, and therefore there are no checks to see if a player is cheating. This has the potential to be problematic, but we can assume all players are virtuous.

Network protocols are tightly sealed and very well-defined, but there is no attempt to measure or compensate for network latency. If this game were to be played across the country, or across the world, network latency may become an issue for disqualifications based on time (especially given the current .5 second limit).

There were murmurings of implementing a text-to-speech observer client that would narrate the game for a player at their computer, which could be useful both for the automatic (computer client) and the natural (human client) play style. Along with this, a human-friendly GUI client would be developed, so that humans could play by clicking on cards as opposed to typing the actual commands into a telnet client. We have found the development and play of the game of Vector highly entertaining, and would be interested to play with our newly formed VSE program. However, playing is somewhat painful for a human being, and therefore our playtime is limited.

A User Guide

Welcome to Vector(TM)! Before you begin totally enjoying yourself, there are a few things you must know!

How to Manage a Tournament with the VSE

There are two ways to start the server. If the working station has both Python and PyGame installed, then simply execute:

```
python vector_server_select.py [-g games] [-r rounds] [-t timeout] [-h host]
```

If the working station does NOT have both Python and PyGame installed, then execute:

```
> cd /Project2/build/exe.linux-i686-2.5/
> ./vector_server_select [-g games] [-r rounds] [-t timeout] [-h host]
```

For our concerns, please focus on the second option.

All options in brackets are to be executed without the brackets (i.e. "python vector_server.py -g 10"). An explanation of the options follows:

g	The number of games to play. This must be an integer value greater than 0
r	The number of rounds per game. This must be an integer value greater than 0
t	The amount of time a player is alotted before they time out and are disqualified. This number is entered in seconds. This must be a floating-point number greater than 0.
h	By default, the server will listen on 'localhost'. Note that this is only acceptable when all the clients are to be run on the same machine the server is running on. If you wish to have your server listen to outside machines you must pass this the address at which the clients can reach your computer.

After running the server, it will listen for connections and once it detects 4 players have connected it will launch the board window. The tournament will begin immediately and you will see, in real time, the games play out. Note

that the moves may display faster than you can understand them. After the tournament has finished, the display will switch into "replay mode" in which you can see, play by play, every game in the tournament.

Replay mode is controlled rather intuitively with the arrow keys.

Pushing the up/down keys will display the next/previous game, respectively.

Pushing the left/right keys will display the next/previous round in a game, respectively.

As the game progresses or you step through the replay, the sidebar located on the right of the window will update with the scores at the current moment.

How to Interface a Player with the VSE

Creating the player to interface with the VSE will, in a way, be language-dependent. However, an example follows for Python. Please see Appendix E for the file.

Writing the Player

First, it is imperative to import the following libraries:

```
import random, os, sys, thread, time, getopt
from socket import *
from select import select
```

There must be a function called readline(socket) that takes in a socket as an argument. This function reads in a line and parses for a carriage return (a `\r`). Note that simply by following the protocol, reading in lines and parsing them correctly, it is very easy to set up a "dummy" player. This is most noticeable in the function `lobby_wait` in Appendix E. Following the protocol becomes a matter of `if-then-else` statements.

Additionally, there must be two functions to send the direction and magnitude to the server. In our example, these are `send_direction(sock)` and `send_magnitude(sock)`, which both take in a socket as an argument and send the direction (magnitude, respectively), to the server. The direction North has value 0, while Northwest has value 7; magnitude must be between 0 and 3, inclusive.

Running the Player

Consider an agent (a player) written in Python, which has socket capabilities programmed in, and stored in the file `computer_player.py`. Once the server is running, then simply execute that player code. The following example code shows how:

```
python computer_player.py <HOST> <PORT> <NAME>
```

Note that all three arguments are required! Thus, the agents must be designed to take in (at least) three arguments. Of course, if the agent wishes to do its own internal book-keeping, it may request more arguments, but the VSE requires each player to send the host, port and its name, in that order.

The `<HOST>` argument must be the same host specified to the server upon start-up. Since `HOST` defaults to `localhost` if no host is specified when starting the VSE, then in this case the `HOST` passed to `computer_player.py` MUST be `localhost`. If connecting across a network, then the IP address of the computer running the server must be given as `HOST`.

HOST	This is the host to which the socket must connect. This is either `localhost` or the IP address
PORT	By default in the server, the port is 1337
NAME	The desired name for the current player.

With these instructions in hand, you can now begin enjoying the fast-paced, high-octane thriller that is VECTOR.

B Project 3 Description

N.B.: There should be a separate ProjectThree file description as a PDF.

The Back Story

It was so weird. You and some friends were having fun, playing your favorite board game (Vector) and listening to your favorite LP, when BAM! , you all magically find yourselves 40 years in the future in the year 2009. You have missed 38 years and are totally lost; your clothes are all out of fashion, and very few people know of Vector: essentially, nothing is the same. You and your friends console yourselves by traveling the world and playing Vector (although finding a copy took a very long time).

But then you get word of this Vector tournament being held in Rochester, NY, except with a twist: rather than human players, this tournament consists of *computer* players. You and your friends get all excited; it does not matter that you have to learn how to program: Vector is worth it.

You enter the tournament and are presented with all the necessities to get you started: the tech specs of the server system (cleverly named VSE for Vector Server Engine), the protocol, the rules, and even a very nice user and programming guide. Luckily, the rules are the same as in the original game, so you do not have to learn those.

However, you do have to learn a programming language, sockets and network connections. You are a bit restricted regarding the sockets and the network, but you can (theoretically) use any programming language you want, as long as it supports sockets. To help you out, the team who created the VSE also provided "dummy" players, in what they consider three of the languages most likely to be used: Python, Java and C. This team figures that these examples should help novice network programmers get acquainted with the system.

Some More Specifics

The team who created the VSE is actually a group of undergraduate students at the University of Rochester, and the Department of Computer Science has been so gracious to offer use of their lab machines to host the tournament. Thus, each player get his (or her) own CSUG Dell Machine; that is, each player is limited to the computing power of ONE CSUG Dell machine. You may not hijack other computers/resources. The VSE will run on a separate CSUG Dell Machine, and each player must connect to the server on that computer. However, as soon as the game begins, there must be NO communication between the players: the

server handles any and all communication!

The preceding point is very important. There can be NO communication between players once the tournament begins! This includes players on the same team! Everyone is expected to be virtuous. It would be highly immoral to secretly communicate between players.

The tournament begins once all four players connect. At the start of the tournament, the server randomly assigns teams for all 13 games. Each game has 12 rounds (a round consists of every player picking a direction and a magnitude). For some added challenge, each player may have no more than 0.5 seconds to choose their direction and magnitude. If a player does not abide by this time-limit, that player and that player's team are disqualified from the current game. When a player is disqualified, the game ends and the tournament server moves on to the next game.

Following the protocol is very important: if a player gives malformed input, that player's *team* is disqualified from the game (see immediately above for what happens in a disqualification).

As mentioned before, the rules for the tournament are exactly those of the rules for the original Vector game. The winning team (of a game) receives 1 point, the losing team (of a game) receives 0 points, and in the event of a tie, everyone receives 0.5 points.

The user guide and protocol can be found as Appendices A and D, respectively, in this paper. The README can be found as an associated file submitted along with this paper.

C Sample Image Files

Here we have included some of the pre-rendered images. All pre-rendered images were created in FireWorks. The board is shown in Figure 2, and the cards, vector piece and an arrow are shown in Figure 3. Using FireWorks was very nice due to the high-quality images it produces and the ease with which the images could be created in FireWorks. Because FireWorks uses ActionScript, the VSE has the potential to interface with FireWorks, and dynamically create *any* Vector-style board, provided the ASCII board representation is provided. However, that was beyond the scope of the project and we did not pursue dynamic creation.

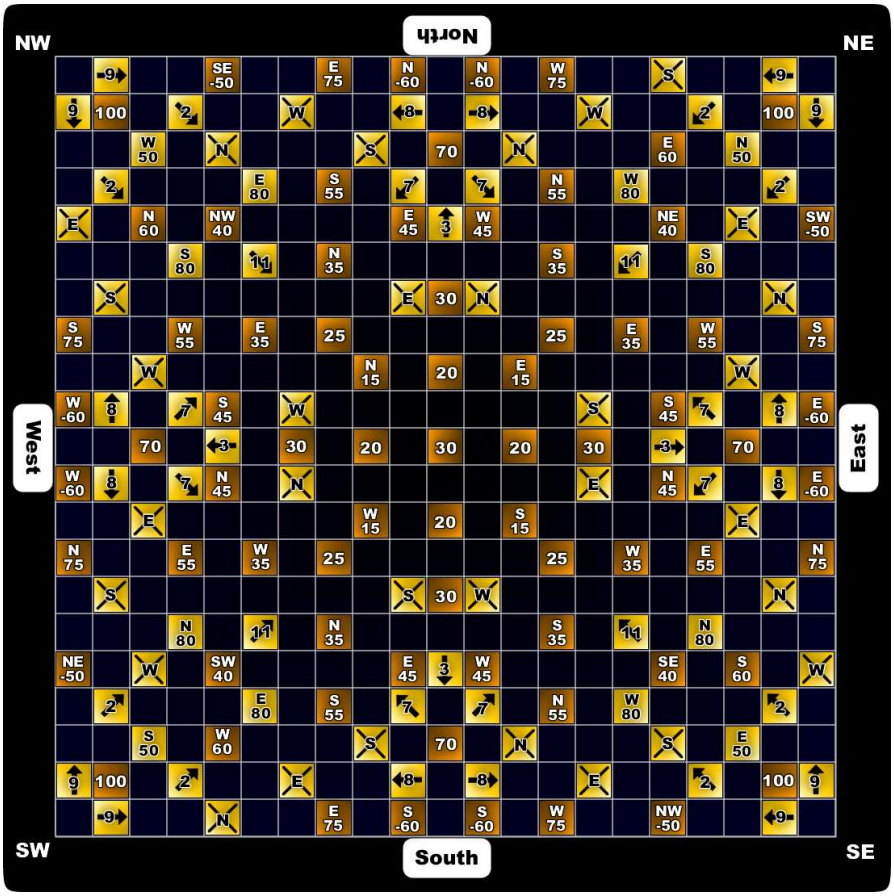

Figure 2: The pre-rendered board.

Note that any loss of resolution or deformities in the images is due to the compression of the image file to encapsulated PostScript. The actual images show all the required detail.

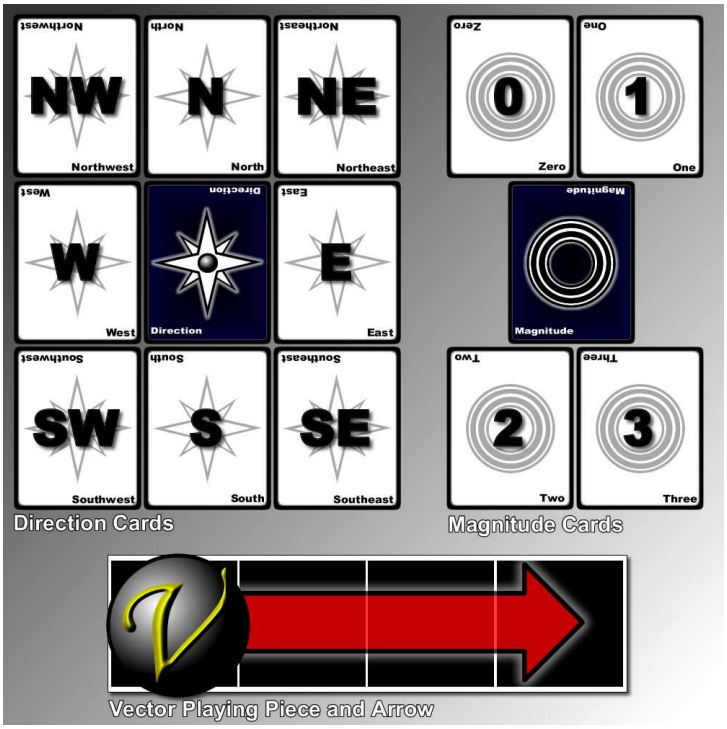

Figure 3: The cards, vector piece and a sample arrow used.

D Unabridged Protocol

First, all clients are asked to identify with a request:

IDENTIFY?

Then, after all have identified, the game turn orders are decided and printed in the format:

GAME <number>: <north player> <east> <south> <west>

After which the first game starts. Each game starts with:

BEGIN GAME <game number> <north player> <e> <s> <w>

Each game starts with:

BEGIN ROUND <round>
COORD: <row> <column>

TURN ORDER: <north player> <e> <s> <w>

When it comes time for a given client to state its direction, it is asked:

DIRECTION?

To which it replies:

DIRECTION: <one of N, NE, E, SE, S, SW, W, NW> \r\n

The server broadcasts the choice of the current player immediately after each player responds with its direction in the form:

<one of N, E, S, W> CHOOSES: <one of N, NE, E, SE, S, SW, W, NW>

Similarly, when magnitude is requested, the client is asked:

MAGNITUDE?

To which it replies:

MAGNITUDE: <magnitude> \r\n

Then, the magnitude choices of all the players is given in the following form at the same time:

MAGNITUDES: <magnitude of north> <e> <s> <w>

After each round, the scores are announced in the form:

SCORES(NESW): <north score> <e> <s> <w>

Should a player be disqualified at any time, their disqualification will be announced:

DISQUALIFICATION: <first disqualified player (identified by N, E, S or W)> <second disqualified player> ...

After a disqualification or the end of a round, the next round starts in a similar fashion. The same holds for each game as well.

At the end of each game, the game winners and the current tournament scores are announced as follows:

WINNER: <winning team or tie, should the game be a tie>
TOURNAMENT SCORES: <player 1>=<score> <player 2>=<score> ...

After a game has ended, either a new game begins or the program finishes.

E computer_player.py

```
import random, os, sys, thread, time, getopt
from socket import *
from select import select
#from engine import *

# random computer player

BUFSIZE = 1024
buffer = ""
#read a line from the socket
def readline(sock):
        global buffer
        index = buffer.find('\r')
        while index < 0:
                buffer += sock.recv(BUFSIZE)
                index = buffer.find('\r')
        line = buffer[0:index]
        buffer = buffer[index+2:]
        return line

#the game hasn't started yet, wait until it starts!
def lobby_wait(sock,identity):
        counter = 0
        while 1:
                line = readline(sock)
                # if the player gets the
                # confirmation that the game started...
                counter += 1
                print str(counter) + "] " + line
                if (line == "IDENTIFY?"):
                        print "PRESENTING IDENTIFICATION TO SERVER"
                        sock.send(identity + "\r\n")
                        print "SENT " + identity
                        play_game(sock)

def sendline(sock, line):
        sock.send(line + "\r\n")

def play_game(sock):
        while 1:
                #get input from server
                line = readline(sock)
                # is it time to send direction?
```

```
                if (line != ""):
                        print line
                if (line == "DIRECTION?"):
                        print "PRESENTING DIRECTION TO SERVER"
                        send_direction(sock)
                #is it time to send magnitude?
                elif (line == "MAGNITUDE?"):
                        print "PRESENTING MAGNITUDE TO SERVER"
                        send_magnitude(sock)
                #elif (line.count("TOURNAMENT SCORES:") > 0):
                elif (line == "TOURNAMENT OVER"):
                        print "TOURNAMENT OVER, DISCONNECTING"
                        sock.close()
                        sys.exit(0)
                elif (line == ""):
                        print "GAME OVER MAN GAME OVER"
                        sock.close()
                        sys.exit(0)
         #if we get a game over, then the client will automatically disconnect

def send_direction(sock):
        directions = ["N", "NE", "E", "SE", "S", "SW", "W", "NW"]
        randDir = random.randint(0,7)
        direction = "DIRECTION: " + directions[randDir] + "\r\n"
        #some code that would send the direction to the server
        sock.send(direction)

def send_magnitude(sock):
        magnitude = "MAGNITUDE: " + str(random.randint(0,3)) + "\r\n"
        #some code that would send the magnitude to the server
        sock.send(magnitude)

# check arguments inputted
if (len(sys.argv) != 4):
        #print 'arguments needed: <address> <port> <playerID>'
        HOST = 'localhost'
        PORT = 1337
        ADDR = (HOST, PORT)
        PLID = "CPU" + str(random.randint(0,99))
        myHOST = 'localhost'
        myPORT = 1337 + random.randint(1,100)
        myADDR = (myHOST, myPORT)
else:
        HOST = sys.argv[1]
        PORT = int(sys.argv[2])
        ADDR = (HOST, PORT)
```

```
        PLID = sys.argv[3]
        myHOST = 'localhost'
        myPORT = 1337 + random.randint(1,100)
        myADDR = (myHOST, myPORT)

tsock = socket(AF_INET, SOCK_STREAM)
tsock.connect(ADDR)
lobby_wait(tsock,PLID)
```

References

[1] Lutz, Mark. *Programming Python, Second Edition.* California: O'Reilly, 2001.

[2] "Vector Instructions". New York: Plan B. Corporation, 1971.