CSCI 265 Proof of Concept

Team name: We Be Daves

Project/product name: See a Neevle, Hear a Neevle

Contact person

- Dave Narealdave, nareal@somewhere.ca

1. Technical challenges.

In this section we outline the four key technical hurdles the team expects to encounter when trying to complete the project:

- multiplayer networking (in Python)
- use of threading, with queues for communication (in Python)
- display of the game screen (in Python Turtle)
- combining Python Turtle with threading.

Each of the four is listed as an issue because they are all areas in which the team has little or no prior applied programming experience, though the third point (display of the game screen) appears to be a relatively lesser risk than the others. Each of the four is discussed individually below:

- **Multiplayer networking (in Python)** involves a host running/controlling a game and other players connecting to the host and exchanging messages as gameplay progresses.
- **Threading and queues (in Python)** involves having queues of messages (messages to send, received messages, game update messages, and display update messages) used to coordinate information across multiple threads running simultaneously (threads for sending,receiving, display updates, and game updates).
- **Game display (in Python Turtle)** involves real-time control of the player game screen and display menus.
- **Threading and Python Turtle** involves ensuring 'turtle' can work in cooperation with threads/queues.

Our proof-of-concept is actually four mini-proofs, tackling each of the four problems above individually. Each of them is discussed (individually) in section 2 and links to the relevant code are provided in section 3. The issue of combining them all together is discussed in section 4.

2. Approach to meet each challenge.

Here we discuss the individual approaches used to show a working approach to each of the four problems in isolation. The issue of combining them all together is discussed in section 4.

2.1 Multiplayer networking (in Python)

The objective here is to create a single program that has both a client mode and a host mode, and that allows a client to establish a connection with a running host and exchange messages.

2.2 Threading and queues (in Python)

The objective here is to create a program that establishes a data queue and at least two threads, and shows that the queue content can be accessed and manipulated by the threads.

2.3 Game display (in Python Turtle)

The objective here is to create a program that sets up a visual display (using Turtle) and draws moderately complex elements on that display.

2.4 Threading and Python Turtle

The objective here is to create a program that combines Turtle with threads and queues in Python, demonstrating that the three can genuinely be used together.

3. Assets produced.

A ProofConcept branch has been created for the project, and within that a ProofConcept directory has been created in the top level of the repository. Within the ProofConcept directory we have created four subdirectories, one for each of the proofs detailed below. Each of these changes has been documented in the standards.md file in our docs.

3.1 Multiplayer networking (in Python)

The implementation for this proof of concept can be found in the Multiplayer subdirectory within ProofConcept, and the executable file is named multiplayerPoC.py. Currently the runtime behaviour is as follows:

- it imports socket
- it asks whether you wish to run as client or host (a host needs to be running before a client can connect to it)
- it sets up a socket for sending and a socket for receiving (the host, ports, address, timeout, and buffer size are all currently hard-coded)
- the host listens for a connection from a client, whereas the client sends a request for connection
- they exchange single word messages (each sending/receiving on their appropriate socket)
- when either side sends a 'Quit' message they both close their sockets and terminate

When one user runs as host and a second runs as client they can successfully connect and exchange simple words.

3.2 Threading and queues (in Python)

The implementation for this proof of concept can be found in the Multiplayer subdirectory within ProofConcept, and the executable file is named threadedQueue.py. Currently the runtime behaviour is as follows:

- it imports queue and threading
- it creates a grab function that takes a queue as an argument and removes/displays the next message

- it creates a queue (to be shared between two threads)
- it starts two threads, each with access to the grab function and the queue itself
- it starts putting data into the queue, letting the two threads grab data out of it

When run, it successfully shows the two different threads 'simultaneously' interacting with the queue.

3.3 Game display (in Python Turtle)

The implementation for this proof of concept can be found in the Multiplayer subdirectory within ProofConcept, and the executable file is named gameDisplay.py. Currently the runtime behaviour is as follows:

- it imports turtle and math
- it draws a fibonnaci-based grid of squares, spiraling outward
- it draws a fibonacci-based spiral, like a snail shell, superimposed on the grid

The program successfully renders the desired scene, procedurally generated, hopefully representing display control as complex as any individual element we are likely to encounter during development.

3.4 Threading and Python Turtle

The implementation for this proof of concept can be found in the Multiplayer subdirectory within ProofConcept, and the executable file is named threadedTurtle.py. Currently the runtime behaviour is as follows:

- it imports queue, threading, and turtle
- it creates two functions, one each for two items to be controlled onscreen (their actions are simply to keep moving, and eventually exit when clicked)
- it creates a function to remove processing commands from the queue and apply them
- it creates the two objects to be controlled and one thread for each
- it starts both threads and the processing queue.

The program successfully moves the two turtles independently once it starts, applying clicks when they occur.

4. Results and implications.

The proofs-of-concept above convinced the team that we are able to get multiplayer communication working, get threads and queues to interact, set up a Python Turtle game display, and get Turtle, threads, and queues to interact successfully.

There are a number of limitations on what the proofs-of-concept showed individually, and a significant limitation in terms of whether or not we'll be able to combine them all together. First we'll consider the individual limitations:

- **multiplayer**: Currently the network information is hard-coded and the host was only interacting with a single client, but neither of those are expected to represent significant hurdles moving forward.
- **queues and threads**: It was realized after the fact that we're only enqueuing from the main thread, not from either of the spawned threads, so further testing is required there.
- **turtle**: Our primary concern here is speed. We found that the update speed for the display was observably slow, even for the simple test case created, which may mean this isn't a viable option to keep all players' displays up to

date at a practical gameplay speed. See the further discussion below for plans to monitor and address this.
- **turtle and threads**: No major issues were noted here, but this point might become moot if Turtle turns out to be too slow for our purposes anyway.

The more significant limitation with respect to our concept proof is that they were each applied in isolation. We are not yet positive that they can all be combined successfully (networking with multiple clients plus threads and queues plus Turtle). It does not appear that the interactions should be a substantial problem, but we cannot be certain of that yet.

The slowness of our Turtle implementation for the display is a more immediate concern. If the display updates are visibly slow even in terms of local display, then the chance of having a responsive multiplayer interaction seem remote. Our current backup plan is to fall back on a straight ascii text representation of the display. We are certain this will fall within our technical skills (basic two-d array handling and text output) and meet our update speed requirements, but will mean a significantly more primitive visual appearance.