Group: Concurrent Tanks
Members: Cam, Lucas, and Omar
11/4/19
Concurrent Programming: *Final* Project

Initial Design Plan: For our project, we attempt to build a simple 2-player game of Tanks using Python and 3 relevant packages: Socket, Pickle, and Tkinter. Python's Socket module allows us to perform server-client interactions using TCP Sockets. Pickle works with Socket by packaging Python Classes into Byte data types, which is required by Socket. Tkinter handle's our graphics and display generation for each client given the relevant Map information.

Our current plans involve us using these packages to do a client-server based model, similar to the chat server we built in erlang (but with more complex interaction). The approach we are taking sends packets of data from each client each time the client inputs a command (move, attack, etc). However, updates occur on a step-base system. I.e., the clients are always acting on the same step, allowing our server to send each client a set, deterministic game-state after each request. Specifically, a client cannot proceed to the next step without a server response and a server only sends a response after receiving a message from all connected clients.

Refined Plan: The overall design has not seen significant changes, other than switching from tkinter to pygame as the primary python3 library for implementing the GUI of the game.

Initially, our intention of using the method Tkinter, was to construct a representation of our map and sprites for our tanks. However constructing sprites proved to be complicated in terms of constructing them and displaying them. We could not find a reliable method function that would allow us to convert images into sprites and struggled to come up with a way to construct sprites by combining built in functions in the methods. Other issues with this method involved issues with retrieving a list of objects to display reliably in a graph (on some iterations the graph would display how we wanted it to and other times nothing would display or there would be missing parts of said graph).

The advantages that Pygame provides are easier display opportunities (such as dedicated functions that takes an image and can generate a sprite), and overall and easier interface to learn with more flushed out examples in how to use the method in videos and explanations. We believe this change will not affect our Development Plan and thus everything will proceed as normal.
What follows is a repeat of the initial design document with minor updates to reflect these changes.

Final Refinements: The design of our Client, Server, and GUI interactions have changed. In our refined plan we had created a class called Memory which would contain relevant information (players positions and directions, list of active missiles, game_state --> if the game has ended or not) that would be passed around between the Client, Server, and GUI. For example, game clients send their information in the format of a memory object to the server. The server then sends a message back to each client in the form of a memory object to update a client on the opposing client's position and direction. This was the initial final design idea for Client Server. However, after running into several bugs with this implementation, we decided to create another object called game state, where each client independently sends their own message to the server through the use of our memory object, the server will instead of sending a memory to the clients, it sends a game state object that contains both client information (like list of players, a list of lists for missiles from both clients, the game over state as well as who won as booleans and a list of all the players who are ready to play) to each client. Each client will then call the update function for the GUI (b/c the GUI is local to the client) to update their display.

- ❏ Phase 1: Simple Interaction (Minimal deliverable)
    - ❏ Tank attacks are handled through simple clicks
    - ❏ Primarily setup to test client-server communication and game feasibility
    - ❏ Game State and Initialization handled server-side
        - ❏ Map generation
        - ❏ State-updates
        - ❏ Game completion/victory
    - ❏ Tank movement updates handled server-side
        - ❏ Collision of Tanks against Obstacles handled server-side
        - ❏ Attack hit/miss handled server-side
- ❏ Phase 2: Varied Interaction (mid-tier deliverable)
    - ❏ Tank attacks have a range-based component
    - ❏ Spacebar to hit now allows for a charge effect where a missile goes longer the longer space bar is held (would have a simple cap)
- ❏ Phase 3: Complex Interaction (maximum deliverable)
    - ❏ Implement a new data class called Missile that is packaged and sent with the Tank and Map information to/from the server
    - ❏ Allows for projectile movement and timed collisions on the map

Final Development Plan:

We plan on meeting twice a week primarily used on check-ins and dividing up work as well as working on assigned roles in and outside of meetings.

By November 11th:

- Able to get a python server running
- Testing out how server and clients respond and send messages in a python
- In the process of designing map

By November 18th (refined design due):

- Testing out sprites with PyGame
- Bare-bones version of the game:
    - Test out Constructed py.game display and see how movement of sprites operate in the GUI
    - Begin constructing server
    - Decided to make all movement keyboard based

By November 27th (Before Thanksgiving):

- Complete construction of GUI
    - Testing --> seeing if movements have lag to them
        - Looking at missile interactions with obstacles vs tanks
        - Looking at game over state i.e. does it display you win message, does it give the option to play again
        - Final draft of sprite selection for tanks, missiles, and explosions
    - Begin Constructing server

By December 2nd:

- Finish Server
    - Test if server is able to receive messages from clients and vice versa
- Begin outlining how to glue GUI and Server together
- Begin preparing presentation
    - Powerpoint presentation
        - Outline Work Done
        - Initial Designs and Ideas
        - Challenges, Pitfalls, and Accomplishments
        - Changes Along the Way
        - Demo
    - Sample game demonstration

By December 5th:

- Completed or near completion of Phase 1

- Completed Presentation and Final Practice for Demo
- Begin Write up of Final Report
    - Testing Server-GUI interactions
        - Looking if server unexpectedly crashes
        - How fast tanks are able to move (how long it takes for the server to respond to a client request)
        - If game is successfully able to be loaded again after a game over (i.e. how play again option works with the server)

By December 9th:

- Fully completed Phase 1 of Game working and running with little to no bugs
- Completion of Final Report detailing what works, what doesn't, how our plan has changed since the beginning as well as a summary of the project and a README

Analysis of Outcome:

Our minimum deliverable was a very simple working game with no background, and we have clearly progressed beyond this. We did not have the chance to integrate mouse support for firing missiles due to the time constraints. However, we were able to implement both parts of our Phase 3 deliverable. We were primarily able to complete this project by identifying key, separate areas to develop and working independently on those regions. Then, when we met, we'd share our insights and work on integrating code.

Many bugs we encountered were due to socket errors and being able to restart the game after a completed round. Both were mostly solved through trial-and-error debugging. The best decision we made was to swap from Tkinter to PyGame. This saved much needed time on GUI development. Similarly, using non-blocking sockets vs blocking sockets made our program run with much less latency and lag.

Summary of files:

TankServer.py: This is the server that is constructed to serve game state data to the clients. This handles updates in the form of Memory objects, and sends to the other player an updated State object, representing the current game state. We cut down on information stored on the server by having it only keep track of any new missiles that were created since the last update by either player.

Client.py: Client is where our Client interactions with the GUI and Server as well as GUI code lives. After initial setup and connection with the server, it enters a loop of processing input from the user (keyboard), querying the server for new enemy tank movement and missiles, updating the pygame

sprites, and finally updating the display shown to the user.

Sprites.py: Here we have all our Sprite functionality. We define Players (Tanks), Obstacles on the map, and Missiles shot by the tanks.

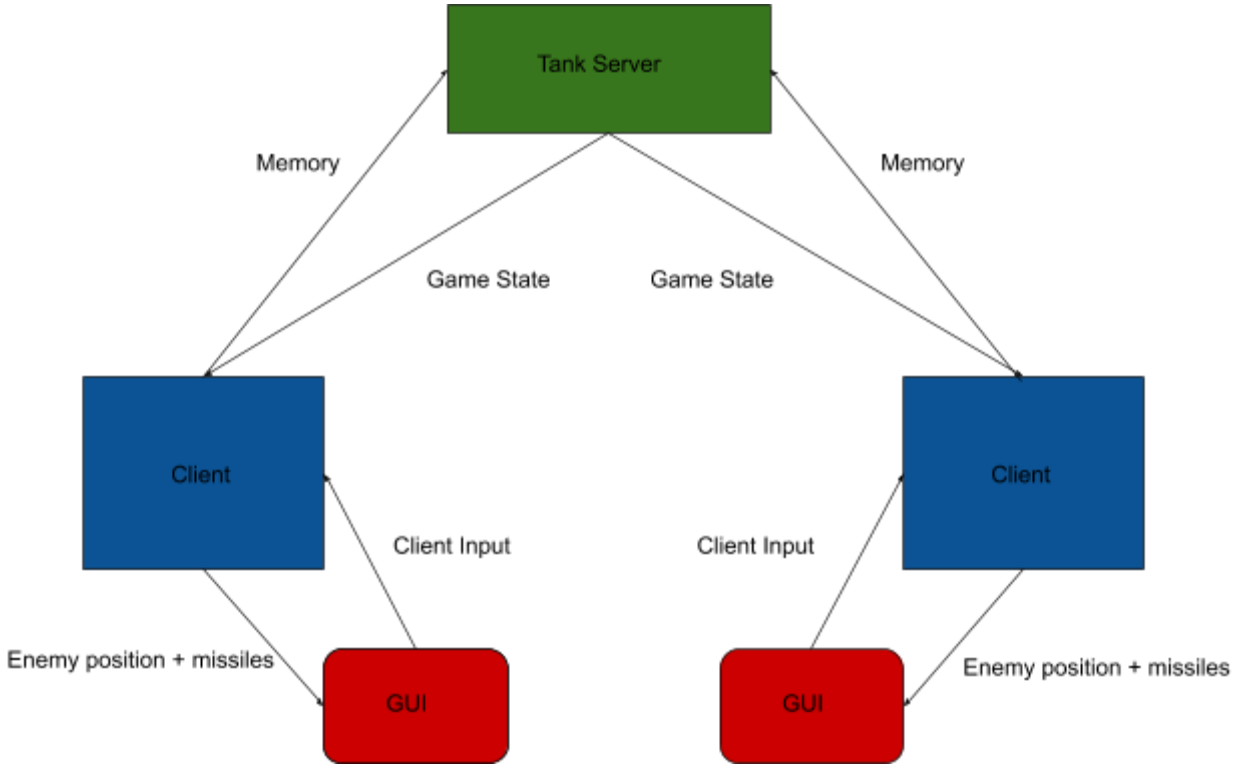
Constants.py: Contains constants used by the various files of the project

Messaging.py: Definition of the Memory, Player_pos, and State classes which serve as structures to hold data passed between the server and clients.

Summary of Project:

Our final project for concurrent programing is a 2-Player Tank game in which each player is a client that connects to a general server. Each player starts on one end of the map, and they are able to move North, South, East, or West using the arrow keys and fire missiles using the spacebar. There are obstacles on the map that provide coverage for the tanks meaning that missiles will explode on contact with a wall (there is no splash damage). Once a player tank has struck the opposing player's tank with a missile, the tank will promptly explode issuing a game over state in which both players are asked if they want to play again.

We establish the connection to the server for both clients with the use of the python socket module and pickle module. Sockets are used to create links between two nodes on a network, using an IP Address. Messages are sent through sockets using the TCP protocol; this ensures that the packets arrive in the order we sent them. After two successful connections have been made to the server, we use Pickle to package our State objects into Bytes objects. On complete message arrival, the server unpacks the Bytes object, updates its copy of the game state, and sends the new state onto the other client. The clients then call an update function to update GUI with the current data (i.e. sprite locations and direction, and missile locations.


Tank Server
Memory
Memory
Game State
Game State
Client
Client
Client Input
Client Input
Enemy position + missiles
Enemy position + missiles
GUI
GUI