Requirements

Written by Sam Young and Aleksander Nowicki

Parent Driver file

`python3 driver.py <Port>` OR `python3 driver.py <IP> <PORT>`

The driver file *shall*:

1. Start from the command line with the form above. If started via the first method, it opens a socket with the given port and listens for connections. If started with the second option, connects to the driver at the given port and IP.
2. Once a connection has been initiated, the driver allows for the selection of serveral options before the initiation of a game.
    - Whether to initiate a game using the delay based protocol or the rollback protocol
    - If the game is using the rollback protocol, how much input delay to have
    - Whether to start a game in the debugging/test mode, which should only display whether the inputs are being correctly captured and sent
3. Once a game has been initiated, the driver starts several threads
    - One to handle the user input, send it via the send method in `server.py` and update the game state accordingly
    - One to update the gamestate at a constant rate based on the local input and the remote input
    - One to listen on the socket for inputs from the remote player, and call methods from `rollback.py` or `delay.py` to handle them
4. From here, the driver shall run until it receives a game over message from the game state handler, printing out a summary of the game and exiting.

Delay Based Netcode Version

`delay_netcode.py`

The delay based netcode *shall*

1. Wait until it has received control inputs over the network for the other player before advancing the frame
2. Not store any history of game state or controller inputs outside of the current frame

Rollback Netcode Version

The rollback netcode version *shall*

1. Receive the remote player's input as usual, and if there are no problems simulate the game state as normal, storing the local input, remote input (if correctly received on time), and game state indexed by frame.
2. If there are packets dropped, then use rollback netcode protocol
    a. Store the last certain frame, and the corresponding game state
    b. Make a primitive guess at the remote input's input, an assumption that the user is still inputting what they were prior
3. Simulate the next game state, using the remote user's guessed input and the local user's input
4. Continue as above until another packet is received.
5. When a delayed packet is received:
    a. Replace the guess of the user's input for that frame
    b. If the replacement is different from the guess, simulate the game state from the point of that replacement up until the current displayed frame
    c. If the newly simulated game state is different from the presumed game state, update it accordingly.
6. Call the necessary methods in the game state handler to update the display of the game for each frame

Core Rollback Data Structure

The rollback system maintains an array with the length of the max frame number, thus each index represents a frame.

Each index in the array contains (GameState, LocalInput, RemoteInput) as a tuple or similar structure

GameState is a list summary of a GameState class
LocalInput and RemoteInput are strings indicating T/F for the state of each of the players' three controls (eg. TTF)

We also maintain two integers which point to indices within the array
last_frame_received = The latest frame for which we have received the remote player's inputs over the network

current_display_frame = The frame currently being displayed to the local player

All GameStates after the one calculated using the inputs at index last_frame_received are guesses using the inputs from the local player at each frame and guesses for the remote player's inputs, as described above.

When a new set of inputs is received from the remote player, these replace the guesses in the array, and each game state up to the index indicated by current_display_frame is recalculated using these new inputs along with new guesses for later frame inputs.

Game Logic Handler

The game logic *shall*

1. Implement a basic, proof-of-concept two-player fighting game with movement, multi-frame attack animations, a health-point system and a victory-condition
2. Implement game logic which is deterministic based on only control inputs and does not utilize floating point math to avoid inconsistency between machines
3. Implement game and controller states that can be easily summarized in a list to save, compare and send over the network
4. Render the game using only the previously mentioned game state
5. Be able to simulate several frames of gameplay in the time allotted before the next frame must be displayed

Message Format

The initiation messages follow the format below during the start of the game, sent from the server to the remote host

- `Connected D` for delay based netcode
- `Connected R` for rollback based netcode

At the start of the game, to ensure that the players are synced up to start, the game will have a countdown, which shall be synchronized between the users and communicated in the following message to aid in starting at the same time:

- `Start 3`, which waits for the other player's start 3 before continuing
- `Start 2`, which waits for the other player's start 2 before continuing
- `Start 1`, which waits for the other player's start 1 before continuing

The messages sent over the network during the game follow the format below:

```
+----+----+----+----------------+
|left|rght|attk|framnum         |
+----+----+----+----------------+
```

Where:

- Left, right, and attack are characters that are either T or F that indicate that the user was inputting those for a given frame
- Frame number is sent as a 16 bit integer, as the time of the game will be bounded by a timer of 99 seconds, at 30 frames per second