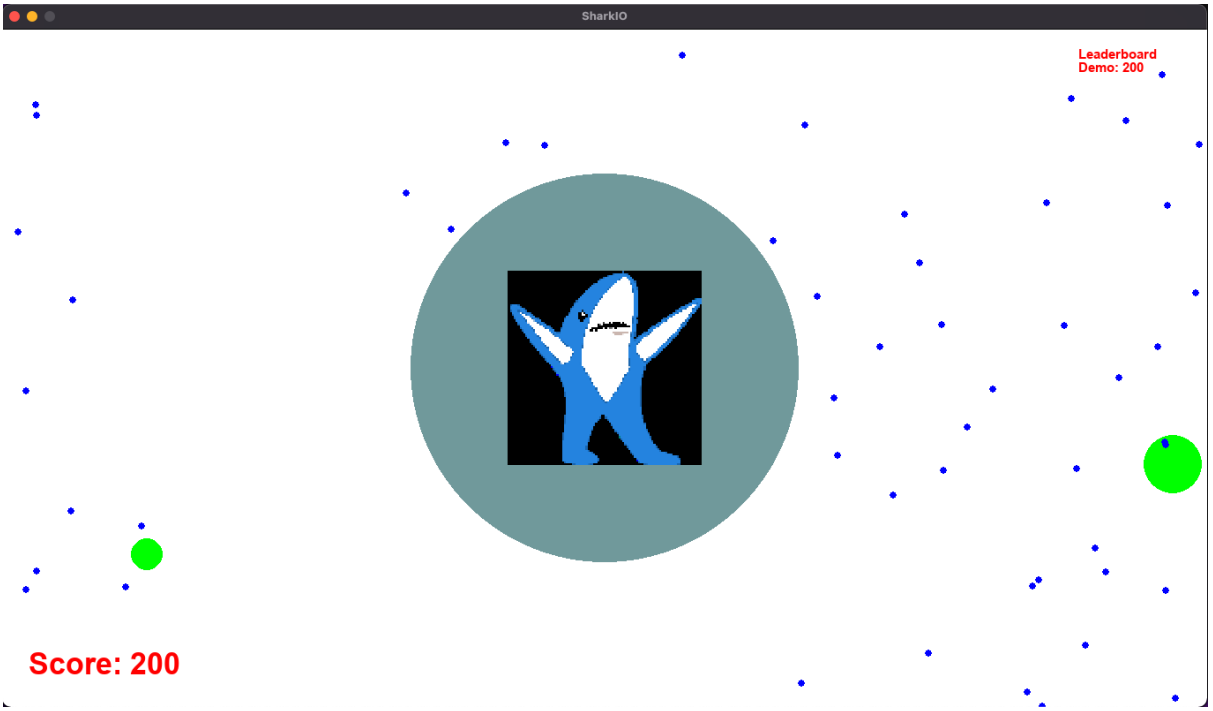
Team Name:

The name of our team is Mark's Sharks.

Team Members:

Our team consists of four members — Ankur Dahal, Ellis Brown, Jackson Parsells, and Rujen Amatya.

Summary:



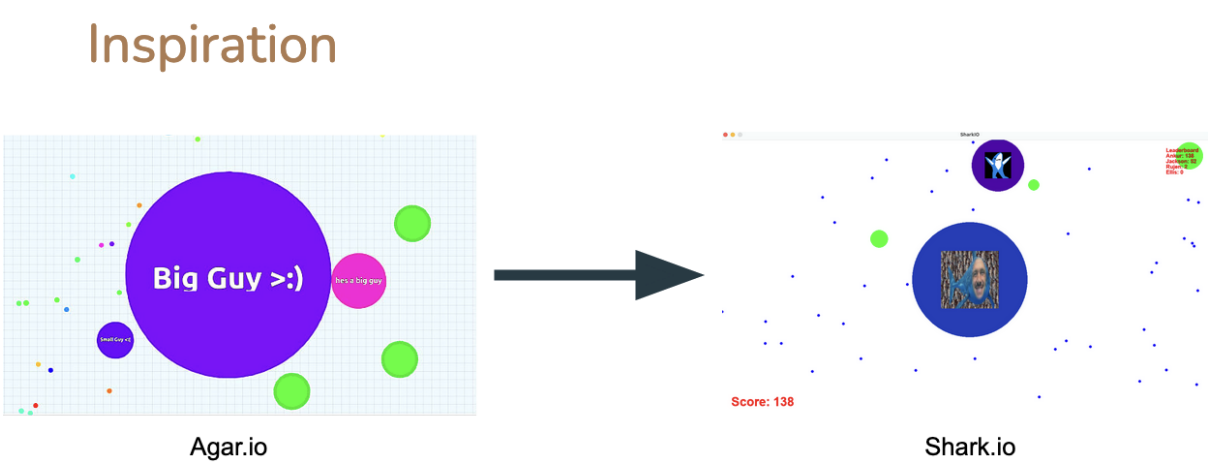
We created a game similar to the popular online games "agar.io" and "snake.io", which are online, browser-based games that allow for 1-20 people to play concurrently. Our game involves moving a blob/shark around, eating food to grow in size, and eating other players to grow in size. A player will control a single blob and can move around in 2-dimensions on the game board. There will be obstacles that the player should avoid in order to not lose their mass. Our game supports 1-20 players running concurrently.

Project Description:

Project: SharkIO

Using Python3, we will be creating an online game with similar game mechanics to the popular online games “agar.io” and “snake.io”, which are online, browser-based games that allow for 1-20 people to play concurrently. Our game involves moving a blob (a “shark”) around, eating food to grow in size, and eating other sharks to grow in size whilst avoiding eating viruses.

A screenshot from the online games that are giving us inspiration for SharkIO can be found here.



A player will control a single blob and can move around in 2-dimensions on the game board. There will be obstacles that the player should avoid in order to not lose their mass.

We are using the following libraries:

- Python client-server model with the client using the **Pygame** library
- **Web sockets** for communication between the server and clients

- **Pickle** for serializing and deserializing data across socket communication

Final refinement of the design, including any changes since the previous iteration and an explanation of why the change occurred.

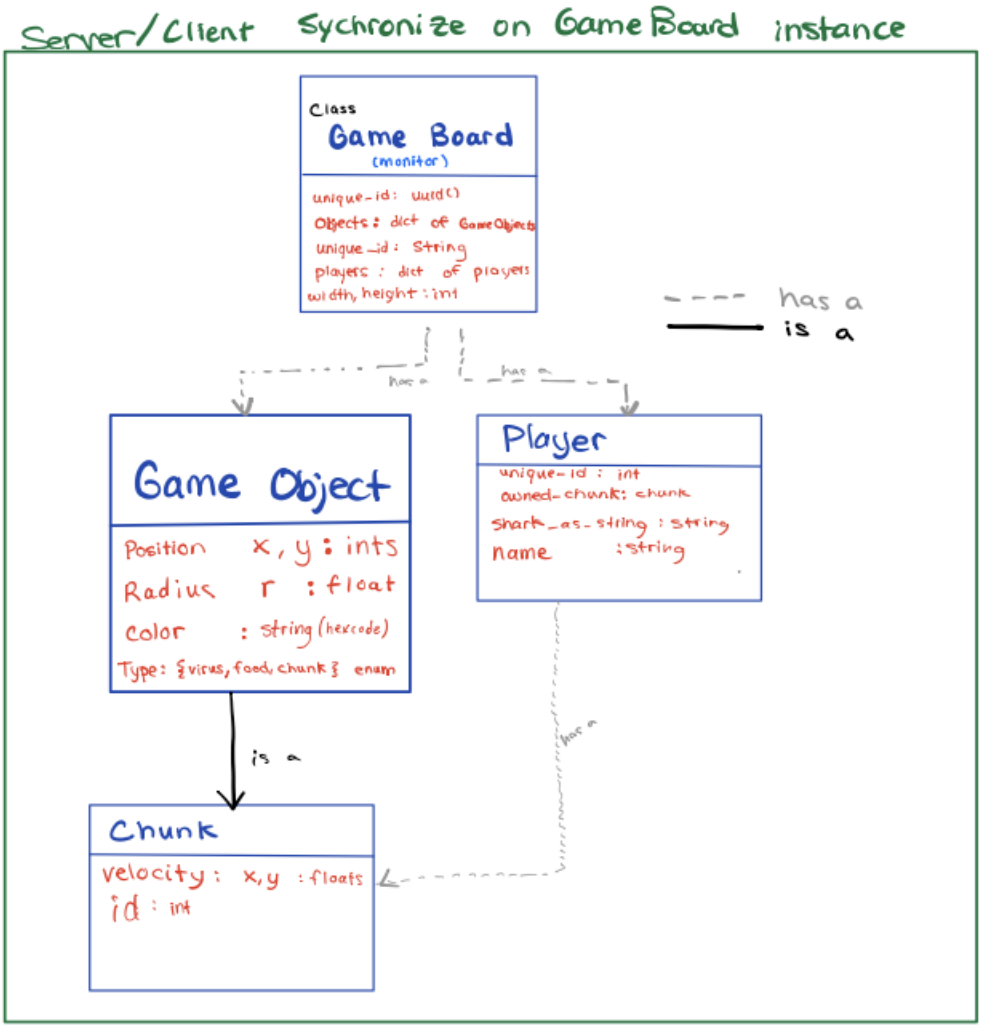
Our design is based on the client-server model, where a server will send relevant game state details to the connected clients and the clients will register player movement and send back the changed position information to the server. On the server side, each client will be represented by their own thread.

The server will have an instance of a GameBoard object, which will contain an array of game objects and an array of players, which shall act as the ground truth of the game state. Each client receives the minimal current game state information from the server to render the game, and then sends player movement changes to the server. The server will handle the collisions and update the required state. All communication between the server and clients is done via web sockets. Each client player instance only renders the view seen by its corresponding window location. Initially, we decided to have a camera class to render each player's view. However, later we decided that we would calculate the coordinates on the client side itself rather than having a camera class. This decision was taken to minimize the size of serialized binary data sent over websockets to mitigate game lag because of large network activity. Another change to our initial design was to have the chunk class inside the players array rather than in the gameobjects array in the server. This change was made because out initial design did not make sense once we started implementing the program, and it made sense to instead make the chunk a part of the player. So, currently, we can explicitly call the gmeboard.get_players() method and on each player, the get_chunk() method to retrieve the chunk for that player in question.

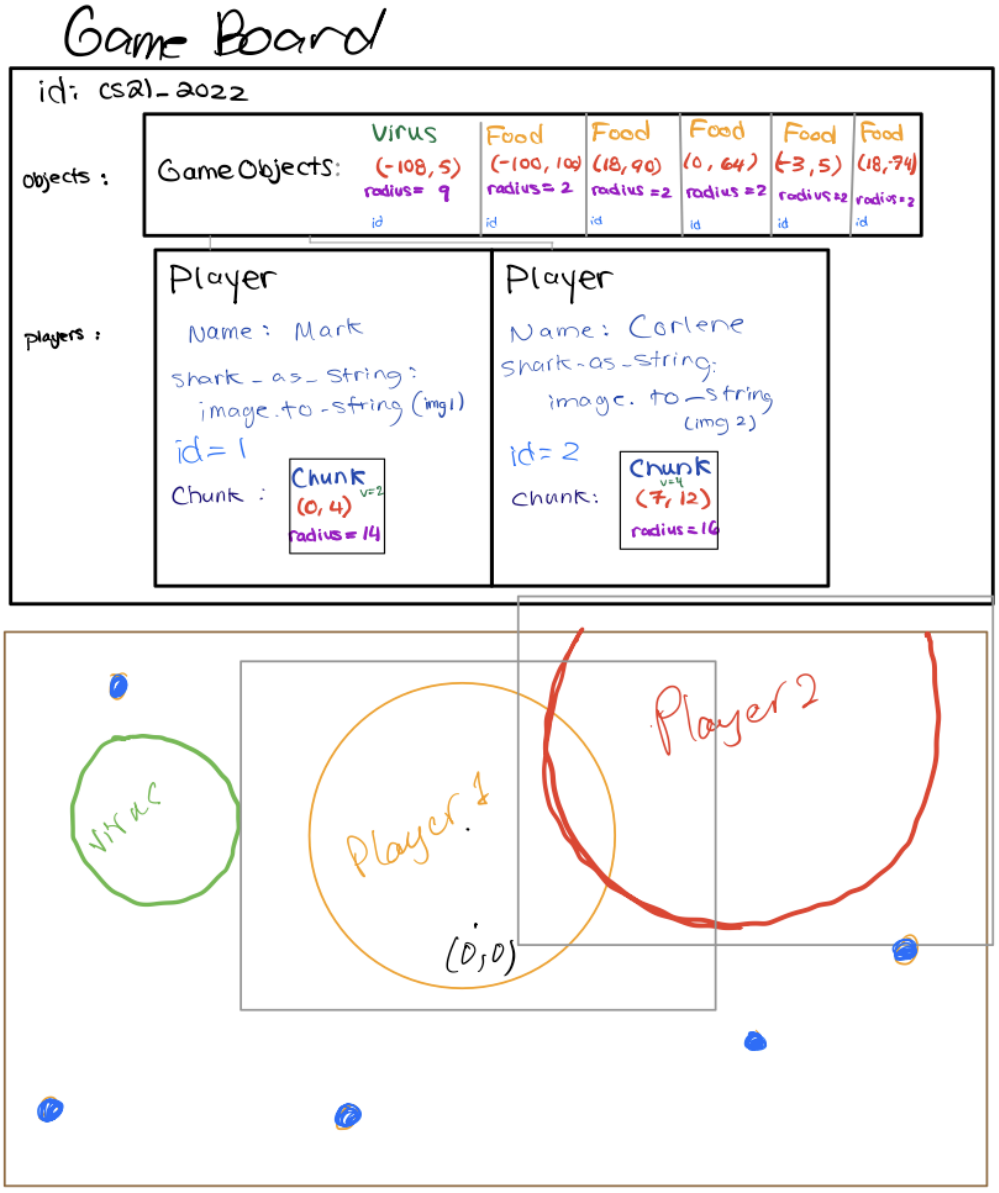
The concurrency portion of the project is seen through multiple threads for clients and a thread for the server. Each client has a process spawned on its local machine, while the server has a thread that represents each client spawned locally, and another thread for tracking the game state overall. Each thread has a one-to-one correspondence with the client process it represents on the server-side. Furthermore, a constant running server thread allows new connections, and another server thread spawns food and obstacles in the background.

Our design decisions evolved as we tried to write our first set of classes. In our initial proposal, we mentioned that we were going to use a Python backend and use a web server to implement our game. However, after analyzing the options, we deviated toward using the Pygame library to make the game because of its easy graphics rendering API, support for sounds, and overall ease of use of the library and the vast documentation available for it.

Class Diagram



- Object diagram:



This is an example of a collision between two characters on an example board. The gray rectangles around the players are the regions of the screen each player can see. Blue dots are food, and the green circle is a virus.

An analysis of the outcome. Did you achieve your minimum deliverable? If not, why not? Did you achieve your maximum deliverable? If so, what went right (that is, why was the project easier to complete than you thought)?

Our minimum deliverable was to implement a single-player game and we planned to focus on just getting the game working, so animation was not part of it and the players would be the same size despite how much they've consumed. We completed our minimum deliverable. Our maximum deliverable was to implement the multiplayer mode, with up to 20 players, and our in-class demonstration would allow every student in the class to play concurrently. We also aimed to add animations and graphics to show the scores of individual players, and also have a global leaderboard to display the current leader in the game room. We successfully implemented the multiplayer mode, added shark animation to the blob, and graphics for the score and global leaderboard. We have also tested 20 players playing concurrently over the network, but decided not to run a demonstration in class, as just running the demo would require everyone to get the source code and run the play script which we thought would go outside our suggested 10 minutes.

A reflection on your design. What was the best decision that you made? What would you do differently next time?

Best design decision: Our method for packing sizes when sending messages over sockets was our best design decision, as it allowed us to wait for the optimal amount of data from the TCP socket each time instead of wasting time rechecking or waiting for too long. Moreover, we thought the design decision for using a player thread on the server side for each client allowed us to write simple and easy to understand concurrent code, which made implementing and testing the concurrent project easier.

What we would do differently: We would not separate chunks from players and instead just have a player class that is inherited from the gameobject class. If we had done this instead, there would be no need to maintain a separate player's dictionary in the gameboard instance on the server-side. Since we had a separate class for a player and a chunk in this design, we would have to do more bookkeeping to ensure that all object instances are correctly represented at all points in time during the game. Furthermore, we would love to extend our game to include movement prediction using interpolation, as demonstrated by another group presentation. We were not aware of such an idea, and did not realize how smooth it could make our game visually.

A reflection on the division of labor. Were you able to divide the project in a way that allowed the members of the team to work independently and then integrate their work into a whole?

We met during class in addition to 2 other synchronous meetings over zoom each week, totaling 6-8 hours per week of labor per person. Therefore, we usually worked together synchronously on each ticket. This enhanced both overall team learning and cohesion. We created tickets at the end of our sessions for each issue so we could easily resume working in future meetings. Occasionally, we created individual branches, worked on the issue, assisted each other to solve it, and merged branches successfully. In doing so, we learned a lot about the importance of version control as well, on top of learning to work together on the same application as a team.

Bug report: what was the most difficult bug you had to find? How did the bug manifest itself? What did you do to find it? How long did it take? What was the cause of the problem? In retrospect, what could you have done to find it faster?

We came across some strange non-deterministic behavior when a player would eat another player and still survive, indicating we had an unhandled data race amongst multiple player threads. We had handled concurrent data races by adding and removing game objects from the state within a thread safe monitor class, but this was a game logic issue due to the persisting values in local thread scopes. We spent a lot of time (>3 hours) considering the collision code, making sure our gameboard monitor was correct, and other things that were not the issue because we assumed our data races were handled incorrectly for adding and removing from the gameboard. However, it turned out to be correct thread-safe data structures, but killed players would still locally exist in their thread's scope, and the data race was adding them back to the board. This led us to add the **state_lock** mutex to player collisions, as we realized the threads independently handling player collisions could have an object killed, but another thread in the loop of updating state would still hold a local copy, which would then be written to the shared gameboard state.

Another issue we encountered was a large amount of latency when playing over the network, as well as pickle sending "truncated data", meaning the server had sent the correct data, but the client did not receive all the data. We spent a lot of time considering the network as the source of our problem, since locally (client and server on localhost), everything was fine. We had not derived the source of the problem when we looked online for suggestions and found one that worked. We followed a stack overflow suggestion which claimed that not all data was being received in its entirety over the network (via websockets). The post (credited in the code block in our submission) suggested that, rather than taking turns communicating with a single message, the server thread / client would accept multiple incoming messages until all of the data had been received. For example, the server now would send X messages until all data was received, then the server would listen while the

client sends Y messages. The length of the messages was padded to the original sent data packet, so the receiver would know the number of expected incoming bytes. While this stopped the truncation, lag was still occurring, so we investigated changing the amount of data sent as well.

While we had originally been sending the entire game board state, of all virus, player, and food locations, we realized the game did not scale well for many players on a larger board. This manifested itself as lag on the client-side for non-localhost computers. This led us to believe it was sending data over the network, rather than game logic, as any client on the same computer as the server was very smooth. We instead had the player thread on the server-side calculate the minimum data required to render just a single frame of the game state, and send that data instead. The window view is MUCH smaller than the gameboard (Window is 1388x768, and the default gameboard is 3000x3000), the amount of data drastically decreased, and our lag as well.

An overview of your code; that is, for each file, please describe the contents of the file and how it relates to the other files.

An overview of relevant files in our project is given below:

- `setup.sh`: A bash script that installs all required dependencies.
- `requirements.txt`: A text file listing python dependencies used by the project.
- `server.sh`: A bash script that starts the server.
- `play.sh`: A bash script that joins the server and starts the game on the client's side.
- `README.md`: README file for SharkIO; contains information about how to play the game and other interesting details about the project files.
- `shark_images`: A directory containing eight shark images (pngs) that are rendered on top of player blobs when they join the game.
- `src`: A directory that contains the python source files for SharkIO. The following files are present:
    - Source files defining classes:
        - `chunk.py`: This file defines a `Chunk` class, which inherits from the `GameObject` class (described below). Each player has its own chunk, and the chunk stores data about the player's position, current score, velocity, etc.
        - `player.py`: This file defines a `Player` class, an instance of which represents a player and contains data about its avatar (i.e., its shark image), and the chunk (instance of the `Chunk` class) associated with the player.
        - `gameobject.py`: This file defines a `GameObject` class, an instance of which represents a drawable object that can either be a player's chunk (instance of the `Chunk` class), a virus, or a food. It stores data about the game object, including its radius, color, and their unique identifier.

        - `gameboard.py`: This file defines a `GameBoard` class, an instance of which represents a game board. It stores data about the overall game state, including the players (instances of the `Player` class) and game objects (instances of the `GameObject` class of type food and virus) present on the map and their locations.
    - Other source files that interact with the classes and help run the game:
        - `constants.py`: This file defines constants used throughout the project. Source files that require certain constants will import them as necessary from this file. Having a separate file for constants ensures that we do not get multiple copies of constants that are different from one another.
        - `server.py`: This file is mainly responsible for opening and listening for connections actively on its main thread via websockets, and spawning client threads once it detects that a client has asked to join the server. One `GameBoard` instance is created and the game begins: each client will have its own thread running (until the client disconnects) which will perform collision detection and update the gameboard state as necessary, and communicate the changes to the state to the connected client via websockets.
        - `client.py`: This file is responsible for establishing a connection to the server via websockets, receiving the gameboard state data, rendering the required output on the client's window using the pygame module, and communicating the player's position change to the server via websockets.

Instructions on how to run your program.

Follow these instructions to run the program:

Ensure that you are running python version 3.9 or later. To create a python virtual environment and install required dependencies, run the setup script using `./setup.sh`. This script is present in the root directory of the project; make sure you execute it from the root directory.
Note: If you see an error regarding installing the packages, you need to manually install them by using `pip3 install -r requirements.txt`.
Once the required packages are installed, it is now time to run the game.
Note: Since this game uses a client-server model, you need to connect to a host running the server in order to join the lobby and start playing. Open the `src/constants.py` file, and change the HOST constant to your public ip if you want to play with other devices connected on the same network. If you want to join as multiple players from your own laptop and not from other systems, you can leave the HOST as localhost ("127.0.0.1"). If any error related to "Address already in use" is seen, change the PORT constant in the `src/constants.py` file and try running the server again.

To run the server, go to the root project directory and execute the server script using `./server.sh`. If there are no errors and you see a welcome message from pygame, you should be all set! Open up another terminal window and navigate to the project's root directory, and join the server as a shark by executing the play script: `./play.sh`. Try joining the server from multiple clients (but remember! They must have changed their HOST and PORT constants to match what the server has in order to connect successfully) and have fun!

The controls are simple: once you have entered the lobby and see your tiny shark floating in the sea, use [W A S D] keys to navigate around the map. That's it! If you prefer arrow keys to move, you can use them instead. The main thing is to enjoy the game and avoid being eaten....right?