Tufts University

Department of Computer Science

Concurrent Programming
Final Project

Team: Meow
Github: https://github.com/slh93/concurrency-final-project/
Branch: Master

Rachel Marison, Sam Heilbron
Friday, December 9, 2016
Mark Sheldon

Project Idea:

This is a game of tag, very similar to agar.io (http://agar.io). Users join a game and they move around an enclosed space as a sphere trying to "eat" others. When they do, they become larger and as a result slower. If you are eaten, you are eliminated and the game is over. To "eat" someone you must overlap his or her center position. There are also small food items sitting around that don't move and allow players to grow easily. There are 2 forms of AI opponents, one type that moves randomly and another that moves towards you. You have 30 seconds to eat all the opponents (AI and Food) or you lose.

Minimum Deliverables:

1. Have the basic game functionality work: Players can move, eat each other and have their speed be affected by their size.
2. Players eat each other by simply running into other players; whichever is the larger player eats the smaller player.
3. Game ends after human player eats everyone else or is eaten

Maximum Deliverables:

1. Allow users to split themselves in half and maintain both halves next to each other.
2. A game clock exists so that the human has a limited time to eat everyone else
3. Make it so that players must completely encompass another player in order to eat them.
4. Create a nicer UI for the game.
5. Create an in game chat at the same time.
6. Create opponents which have varied movements
7. Create process so that the winners of games can play each other.

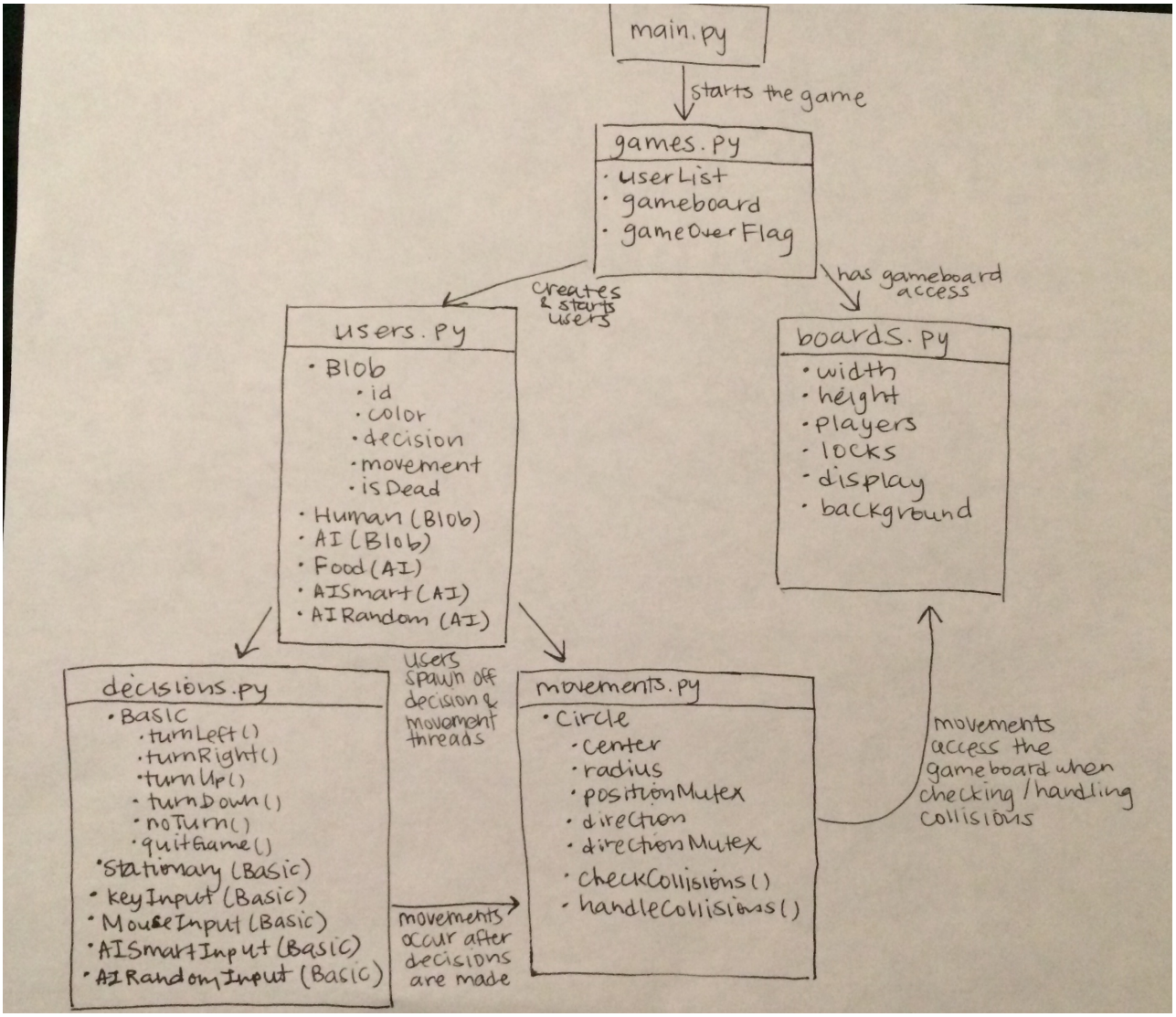
Design:

We decided to use python for this project since most multiplayer games that we had researched are written in python and there do exist helpful libraries.

We broke up the project into smaller components such as decisions movements, users boards and games. The lower level information of each of these components is described in detail in the "Overview of code" section below.

Our design was an attempt to make our project as modular as possible but to also follow certain concurrency decisions that we made. See below in the “Concurrency Decisions” sections below for more information about those decisions.

Since the last report, one of the biggest design changes was the addition of a locks board and player's board to the boards.py file. The locks board is a 2D array of the size of the gameboard, with a lock at each index. It ensures that each location on the gameboard can have at most one user in that position. The lock board also makes it really easy for us to determine when a collision between users has occurred. This easy lookup using the locks table ensures that the UI is not slowed down. The player's board is a 2D array of the size of the game board that stores user IDs by their center positions on the gameboard. This board provides us with easy access to the users on the board. See below for an outline of the class relationships and overall architecture of the game:



Analysis of outcome:

We achieved everything that we listed in our minimum deliverables. We were also able to achieve some things from our maximum deliverables. For example, we were able to come up with a pretty decent UI through the use of pygame, and we were able to make it so

that users must encompass other users in order to eat them. Furthermore, we added a game that timed the user. However, we did not add in functionality for letting users split themselves in half, or for an in-game chat. We also didn't implement multiplayer functionality. In place of multiplayer functionality, we decided to just create several AI computer players that the user would play against. Many more opponents could be created easily because of the modularity of our code. We have successfully decoupled components such as size, decision-making, decision speed, and movement to allow easy flexibility for creating new user types.

Certain parts of the project were easier than we thought. For example, getting the UI to work was pretty straightforward using pygame. One aspect of the project that we thought would be very difficult was dealing with collisions (how to know when a user is being eaten). However, this went very smoothly because our locks board in boards.py allows us to easily detect when two users collide.

There were some challenges that were present as well. We ran into a bug that disappeared only when we changed the size of the pygame display. Users were going out of bounds (outside the dimensions of the 2D array). This was causing and IndexError when the lock was acquired since the lock didn't exist. However, it's not possible that this happen since only the center of users actually acquire the lock and the closest they can make it to the border is their radius away from it. This seemed to flare up when a user ate another by the border but again, the radius would change, not the center as a result. After changing the display size, this error disappeared. We break this down to pygame being brittle because we have account for race conditions (there are none) and edge cases (off by 1 errors).

Reflection of design:

One of the best decisions that we made was to have each user have two threads, one for "decisions", and one for "movements", instead of just having each user run on one thread like we originally planned. The "decisions" class handles the decisions made by the user; for example, the user makes the decision to press the up arrow key when they want to move up. The "movements" class then handles the actual movement of the users based on the decisions. This was a really good design decision because it helped to organize the code and it clearly separated the implementation for registering a user's input (decisions) and processing that input (movements).

If we could do something differently next time, we would've liked to make the game a multiplayer game. I think the main reason we didn't get to implement any multiplayer functionality is because we didn't think about it much during the design process, so it was more realistic to just focus on making it a single-player game. However, if we had more time, we would’ve liked to make it a multiplayer game.

Division of labor:

Since we used GitHub, we were able to divide the project up and work independently. We met up a few times to update each other on our progress and to discuss some specifics, but we were able to work separately and still smoothly integrate our code together.

Bug Report:

The most difficult bug to find was one where users were unable to eat anything on the rightmost border of the boardgame. If you collided with another blob on the right border of the screen, you would just glide over them without eating them. It wouldn't even register as a collision. This was a very confusing bug since we couldn't think of anything that would cause the bug to manifest in just the right border and not the rest of the gameboard. At the moment, we still haven't been able to figure out what the bug is. We've considered possible indexing errors, but that doesn't seem to be the problem. We're still currently trying to fix the bug.

Overview of code:

main.py:

Prints out an opening message explaining the rules of the game to the user. It then prompts the user to select their preferred form of input (mouse or keyboard). Creating and running an instance of the Game class then starts a game.

games.py:

Contains all possible game classes. A Game is a wrapper for all components of a game. It keeps track of the users alive in the game (userList). It also keeps track of the game board, which is described more in boards.py below. It sets up all the users, draws and redraws them on the board, and kills users that have been eaten. It also listens for the game to finish by waiting on the gameOverFlag, which is signaled when the human is eaten, and the gameOverTimeout, which is signaled when the game clock has run out.

boards.py:

Contains all possible board classes. The *SyncGameBoard* class represents a thread-safe game board for the game. It handles creating a pygame display, a board containing the center position of all users, and a board containing locks for each position of the game board, thus allowing for atomic access on the game board. The SyncGameBoard is in charge of moving users from one place to another on the board as well as removing them once they have been eaten.

decisions.py:

Contains all possible decision classes. Decision classes represent the types of decisions that users make in order to move. All classes are built on top of the Basic class, which connects the decision that is made to the movement class (see below).

- *Stationary* class always decides to stay in place.
- *KeyInput* class handles keyboard inputs to change the users direction.
- *MouseInput* class handles mouse movements to change the users direction.
  AISmartInput class decides to move towards where the human player is
- *AIRandomInput* class randomly chooses a direction to go in

movements.py:

Contains all the possible movement classes. In this game, all user blobs are circles and the *Circle_* class handles the actual movement of the users.

The Circle_ class contains attributes related to its position (center, radius) and a semaphore is required to get and set these values. It also contains a direction attribute, which it shares with a decision class and thus another semaphore ensures atomic access to this value. The Circle class tells the board to update with its new position and then is responsible for handling collisions. The Circle looks for locked positions on the game board that are within the radius of the circle.

users.py:

Contains all the possible user classes. All users are built off the base *Blob* class. The Blob class has decision and movement classes (explained above), a color to display to the screen, and id to distinguish them from other blobs and a threading event isDead, which is triggered when the user is eaten. This event kills the threads, which are controlling the users decision, and movement threads.

- *Human* class represents the human user in the game. The movement class is handled in a separate thread but pygame requires that IO be handled in the main thread.
- *AI* class is a base class for all non-human users. Both movement and decision instances are handled in separate threads.
- *Food*, *AISmart*, and *AIRandom* all inherit the AI class and just vary in their attributes.

enums.py:

This file contains all enumerations used in the code. These include *Direction*, *Color*, *InitialUserRaidus*, and *Timeout*.

Instructions on how to run the code:

In order to run our program, you must first download getch and pygame. In our "src" folder, you can find the folder "install", which contains a file called "instructions.txt" which describes how to download getch and pygame.

Once everything is downloaded, you can run the program by going into the "src" folder and simply running "python main.py" or "./main.py".

Concurrency Decisions

- *Users decisions and movements*: Since humans can make a decision with their brain, while also doing something with their body, it makes sense for users of the game to have this capability. Therefore, 2 different threads handle the decision and movement classes. This successfully decouples the rates at which decision and movements are made.
- *User movement*: There are 2 ways to handle moving users around. The first is to have all users wait at a turnstile and once everyone has arrived proceed and make a move. However, since users can move at different speeds, users would have to move more than 1 pixel at a time to account for varying speeds. However, this could create scenarios where users jump over each other because they move so many pixels that they never collide but instead travel through each other. Therefore, we used the alternative to user movement. Each user moves independently of others at an

interval different from others. It is up to the scheduler to fairly allocate resources to each thread. This way, all users travel one pixel per movement.
    - Side Effect: Since the scheduler controls when the context is switched, 2 users may enter their loop to move, the first eats the second, kills it, and removes it from the board. However, the user that just died has already passed through the turnstile and will remain for 1 movement. We add a try...except to catch this case and throw it away. This is a required side effect of how we handle user movement. However, it is a better one than what would have occurred if we handled movement differently.
- *User direction*: Since the ability to decide and move have been decoupled, the two classes now share a resource (the user direction). Therefore, a Semaphore is used to ensure atomic access.
- *User life (and death)*: Since there are 2 threads for each user (except the human), there needs to be a clear way to kill both threads. The isDead event is responsible for this. When it is triggered, both the movement and decision loops will terminate, killing the threads.
- *Board movement*: Multiple users are sharing a board and therefore a 2D array of Semaphores is required to ensure atomic access to a position.
- *Collisions*: When 2 users collide, it must first be determined which one is larger. The smaller one will be terminated. However, one that is about to be killed will continue operating (and changing position) until it has registered to be killed. Therefore, the larger user must block the smaller user from moving; create a kind of turnstile, so that only after the smaller user has been killed will it be released. To do this, a Semaphore is place on the center of all users. The center is not a shared resource (only the movement class changes it) but this allows other classes to acquire and hold the lock.
- *Drawing*: Drawing the board continually is a separate concept and thus handled in its own thread. This allows flexibility with frequency of drawing.
- *User placement (on the board)*: Users are not guaranteed a placement on the board that is separate from all others. The only user guaranteed to not overlap anyone is the human. This was a design decision. As soon as the game begins, the larger blob will eat the smaller one. This variability (based on probability that 2 randomly chosen locations overlap) adds a level of excitement and randomness in the game.
- *Game termination*: There are three ways that the game can end. First, the user may run out of time. Second, a larger AI may eat them. Finally, they may eat everything before the game clock reaches 0, and in doing so win the game. There are 2 cases that must be handled: a collision triggering the human death and a timeout of the game clock. We treat these two cases in separate threads and then create a rendezvous to ensure that both terminate once one terminates.