Jennifer Frase
Zacharia Barnes
Adrian Williams

Zapin Mid Point Check

Running the game: While in flip open up the folder that contains Mid_point_game.py.

1) Start up the server
   a) In its own console window by typing 'python3 server.py'
   b) Or in a window that you will play in by typing 'python3 server.py &'
2) Connect to the server with the host in a unique console window by typing 'python3 midpoint_game2_0.py host' then follow the prompts. Do not change the window size.
3) Create a duplicate session of the host. Connect to the server with the guest by typing 'python3 midpoint_game2_0.py join' then follow the prompts. Do not change the window size.

Playing the game: There will be a start menu where you can choose to start a game, look at the instructions or exit. You navigate the menus by pressing the indicated key to choose the option. As the instructions page (accessible from the start menu) indicate you move the spaceship around by using the arrow keys. To pause the game you press "p". Inside the pause menu you can change back and forth between player 1 and player 2, which control horizontal and vertical movement respectively. You can also exit from the game prematurely from there as well. If the spaceship is hit by an asteroid, then it is game over and you will be kicked back to the start menu.

Currently it is possible to play the game in both fullscreen and the standard size screen, which can be adjusted before starting the game. Since you have more warning about where an asteroid will appear, this means that playing in fullscreen makes the game easier.

Files Submitted:

midpoint_game2_0.py - game used to play in for the mid point check.

button_test.py - includes test code for the menus and player 1 and 2 movement. Some menus have not been incorporated into the game yet.

server.py - Server used to connect guests to a host.

Highlights and improvements Perspective From Input Developer:

Currently it is possible to get to any menu through the push of a button without having to wait. I intend to make sure that this is possible at all times. This insures that a user will never feel "stuck" at any point in the game.This will become very important as we add networking to the game, since at that point users will have to wait for responses from other users. If waiting becomes too much for a user they would then be able to back out and return to the start menu, without waiting for a response.

In addition, the main way to navigate the game is by pressing indicated keys. This has it's pros and cons however. On the positive side because the keys are different on each menu it is not easy to accidentally hit the wrong button and do something undesired. However, at the same time this means that the user must search the key board for the right key. So going forward I am looking to modify menu navigation to to also use arrow keys with some flashing text or "cursor" to indicate the selected option.

I am also aware that some people do not like to use the arrow keys for navigations and prefer to use wasd. In order to accommodate these people I will implement an option that can be selected to enable those keys to work the game as opposed to the arrow keys.

Highlights from Engine Developer perspective:

As the engine developer I would like to draw your attention to the following features of the game that I have been able to implement so far, namely: The game loop, the physics engine, and the "game play difficulty algorithm".

Game Loop: The game loop is at the core of essentially all games and thus is integral to the functioning of any game. The game loop, in essence acts as the heartbeat for the game. Because of these facts, I spent a lot of time researching the theory and mechanics of what a game loop should provide. I settled on a game loop of the following general form (Reference: goo.gl/H5VqRZ):

```
double previous = getCurrentTime();
double lag = 0.0;
while (true)
{
  double current = getCurrentTime();
  double elapsed = current - previous;
  previous = current;
  lag += elapsed;

  processInput();

  while (lag >= MS_PER_UPDATE)
  {
    update();
    lag -= MS_PER_UPDATE;
  }

  render();
}
```

The above implementation of our game loop is ideal because it facilitates the decoupling of the progression of the game (that is to say the “physical state” of the game, i.e. the position and velocity of all of the game objects and their subsequent motion with respect to time) from the time it takes to process user input and render the game, as well as from the processor speed of the underlying hardware that the game is running on (assuming a time step is chosen that is greater than the amount of time required for said update). Thus the speed at which the state (not the visual display) of the game updates is consistent across all platforms the game runs on. By making this be the case, we can assume a constant time interval for every update of the “physical state” of the game. This point and by extension the notion that the game state update rate will be independent of the hardware the game is running on proves crucial for a networked game, because it allows for the game state to progress at the same rate across all machines running the game. Thus without going into all the details, the above game loop is great because: (1) it does not waste CPU cycles on a timer to ensure constant game play (rather it spends those cycles doing useful calculations required by the game) and (2) the loop provides a constant game state update rate across all platforms which greatly simplifies the implementation of the physics engine and more easily allows the game to be networked in a manner that provides a fair and consistent game experience for all players involved.

Physics Engine: The physics engine is important because its responsibility it to update the “physical state” of the game. The physics engine, as I have implemented it, is responsible for updating the position of all of the game objects (besides the player, but if we decide to add inertia to the player a.k.a. the spaceship then the physics engine will update the player as well) as well as performing collision detection. The physics engine does these two tasks by taking the constant time step from set by the game loop as an input and then using that time step in conjunction with equations of motion to calculate the next position of all of the currently active game objects and then once the new position for a game objects has been calculated the physics engine checks to see if the game object has come into contacted the spaceship.

Game Difficulty Algorithm: I implemented an algorithm to increase the difficulty of the game as a function of time. The algorithm increases the difficulty of the game by changing the rate at which objects are spawned with respect to time. Thus, I implemented a piecewise function to control the spawn rate such that as time increase the rate at which objects are spawned increases as well.

Highlights from Network Developer:

1. **Connecting 2 Players to Each Other:**
    a. Currently a player starts the game via the console along with an argument identifying the player as either a host or a guest. The intention is to remove the argument here and to have this selection be part of the game menu. Previously the host and guest code resided in separate files, but now they exist in the game file as separate classes.
    b. The host connects to the server and enters their name which the guest will later use to connect to. The name along with the socket information of the host is added to a list maintained by the server. Currently there's no method to handle duplicate names or the removal of names from the list once the host is no longer waiting for a connection. The host then disconnects from the server and waits for a guest to connect to it.
    c. The guest connects to the server and is supplied with a list of host names waiting for a guest to connect. The guest enters a name, and if it's valid it receives the host's socket information. It then disconnects from the server and connects to the host.
2. **Maintaining Identical Game States:**
    a. During the construction of the HostSocket object in the host's game, the object gets a seed and seeds its game, then sends the seed to the guest's game. During the construction of the GuestSocket object in the guest's game, the object receives the seed and seeds its game with that. Since the random module is used to create the game state, the host and guest game state are identical. And because the host and guest both send and receive inputs in each game loop, the state of their spaceship remains identical.
    b. This method may be insufficient later in development as the complexity of how game objects make decisions increases, and it will be updated if necessary, depending on the actions of the engine developer. If for some reason game objects make different decisions despite having the same seed, the game state from one player may need to be sent to the other player, in which case the code for the secondary player may have to be rewritten such that it merely displays the game and sends/receives input, but no game logic actually occurs there.

Proposed Goals and Changes for final implementation:

1) Improve the game difficulty algorithm to provide a less predictable / more random user experience and increase the “hardness” of the game by not only manipulating the spawn rate of game objects but also by making the initial state of an object manipulatable (spawn objects with varying speeds).

2) Implement game object to game object collision detection with corresponding physics reactions (impulse / momentum / change in velocity …. think Pong).

3) Implement embellishments corresponding to game object collisions.

4) Implement addition sentient game objects.

    a. Implement “artificial intelligence” for these objects (objects that respond to current game state)
    b. Make these objects animated (give objects movement capabilities other than simple rigid body motion)

5) Implement user / player actions and abilities (shields, shooting, bombs, etc).
    a. Possibly make these abilities such that the user acquires these abilities through productive game play in order to combat increasing game difficulty rather than just being given these ability initially.

6) Implement a player score tracking system.

7) Make all commands done via the game menu after starting the game via the console with no arguments.

8) Debug!