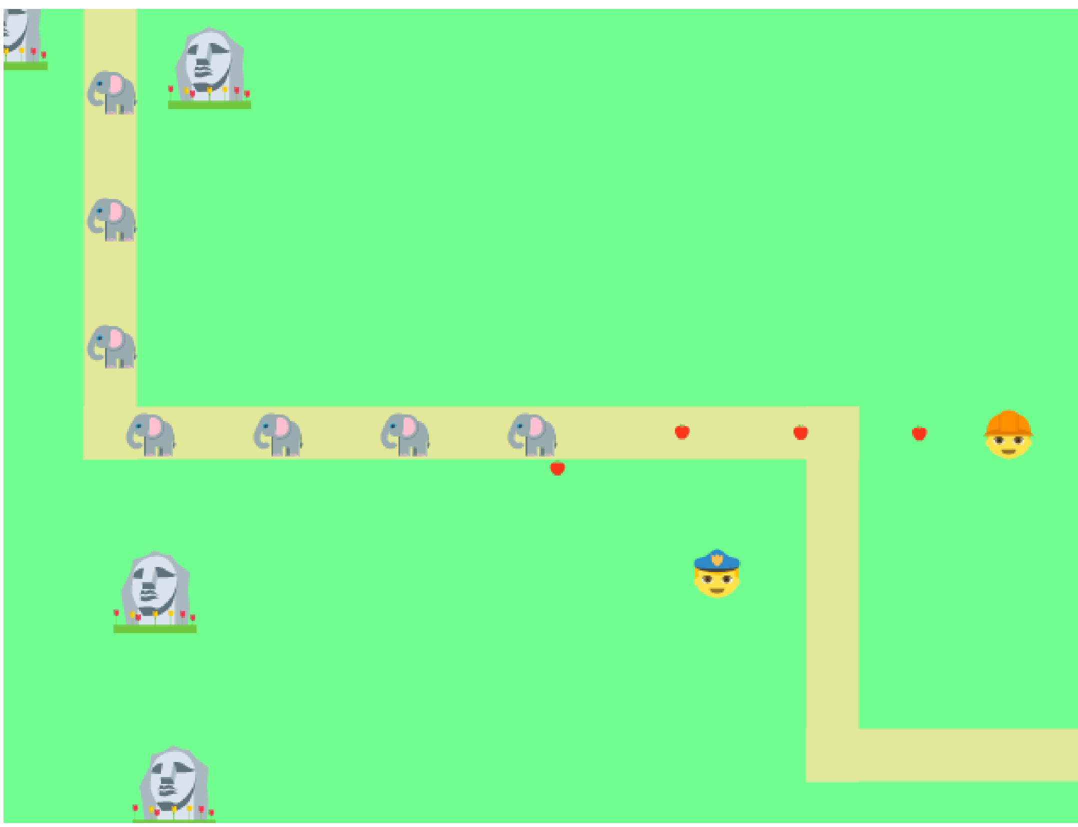
lab9

January 8, 2020

1 Introduction

After a successful stint at MIT, your skills are in global demand! In particular, the world-renowned 6.009 Zoo is in dire need of a manager to help feed its increasingly hungry animal population. They reach out to you and, being the animal lover that you are, you accept. Since you are an avid student of Fordism, you come up with an assembly line process for feeding all of the animals in an efficient and speedy manner. In this lab and the next one, we will build a tower-defense game to model this problem! Here is a picture of what the complete game will look like:



Your game will be played on a two-dimensional grid. On the edges of this grid there will be a start where animals will appear and an end where they will disappear. The animals will move along some defined path connecting these two coordinates. Your job is to feed the animals before they reach the end by placing zookeepers along the path. Zookeepers cost money, which players earn by feeding animals. If a zookeeper sees an animal, it may toss food at the animal, thus feeding it.

To get an idea of how similar tower-defense games work, feel free to play one of the many tower-defense games available online! Tower-defense games are a broad genre, so there are many options. The staff is fond of Bloons Tower Defense.

While a game is hard to describe in full detail, we have restricted our tests to exactly the aspects described on this page. Beyond that, you may make your own design decisions and should be prepared to discuss them during your checkoff. We have presented information in the order which we think maps most intuitively to implementation order, but you need not follow this order. If you find yourself overwhelmed by the many aspects of the game, attempt to break them down into pieces to create a modular design. Have fun!

2 Technical Overview

2.1 Debugging with the Web UI

Using the UI on this lab is highly valuable, and we recommend you read this section in full.

Our web user interface (UI) is a powerful debugging ally and one of the most fun parts of this lab! To use it, run `python3 server.py` and use your web browser to navigate to localhost:8000. You will need to restart `server.py` in order to reload your code if you make changes.

When the server is first started, it calls your render function exactly once. This call to render occurs before any timesteps take place. The UI will display the results of this render. If you wish to begin the server, you can press the "Run" button located below the grid. This calls your timestep function, followed by another render to update the screen. As long as "Run" is selected, the server will continuously call timestep and render in sequence (as fast as it can).

By selecting a test case number from the dropdown, you will be able to visualize the expected results of that test case, alongside your results, using the "Ghost Mode" toggle button. This should be your first resource when debugging.

Feel free to edit `lab9/resources/maps/design-your-own-zoo.json` (or create more similar `.json` files) for more fun. By selecting `design-your-own-zoo.json` as your map on the UI, you will be able to visualize your own zoo. Other files in the `maps` directory are used for testing, so you shouldn't modify values in those.

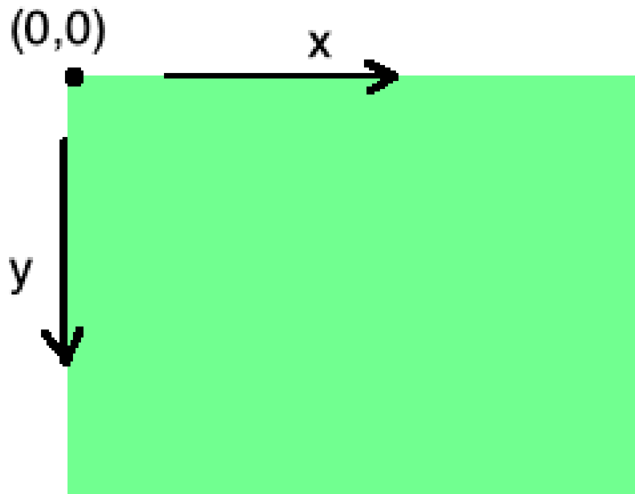
If your code does not seem to be matching with the server, we recommend clearing your browser's cache or opening the UI in a new incognito or private session.

The gif below displays a slowed-down version of a game, with clicks emphasized for clarity.

.GIF DID NOT WORK IN PDF FORM
PLEASE CHECK THE .ZIP FILE FOR CORRECT .GIF FILE

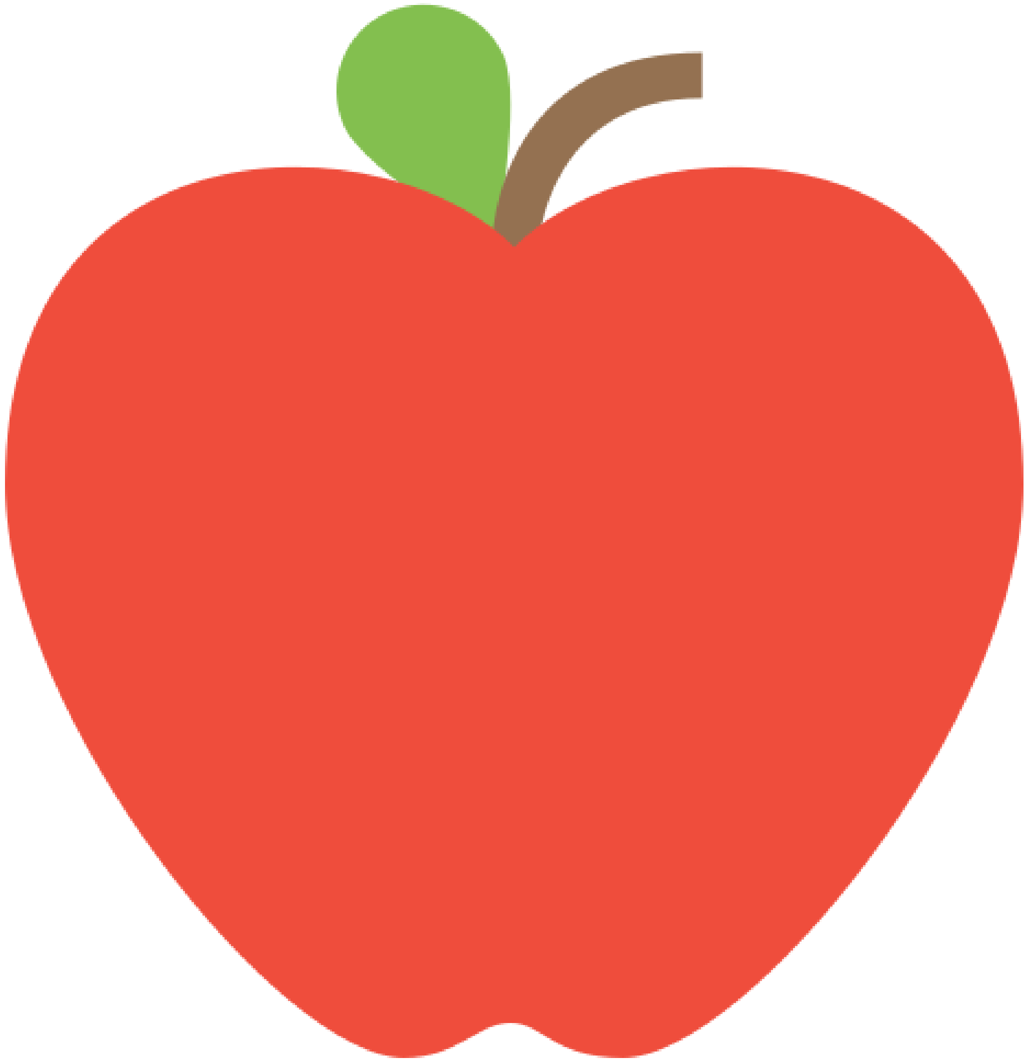
2.2 Grid System

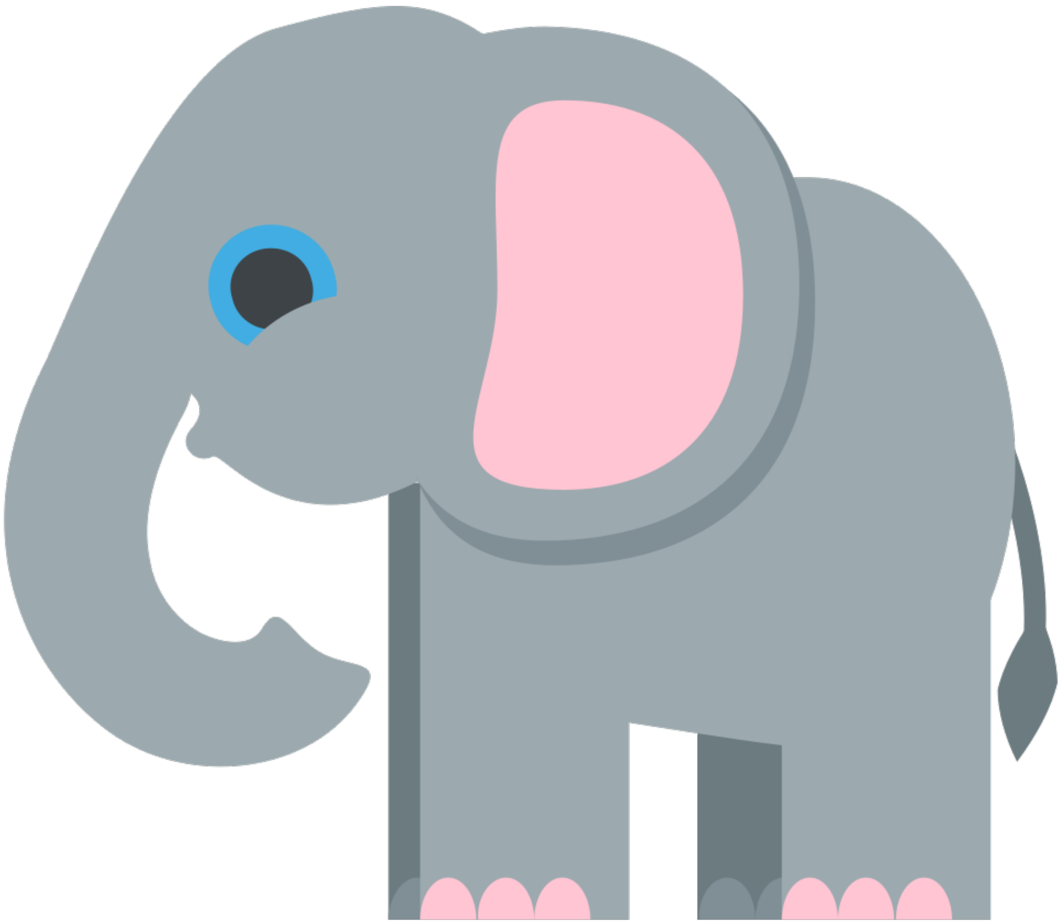
The game world is a two-dimensional coordinate plane. We consider the top-left corner of the grid to be `(0, 0)`, with xs increasing towards the right and ys increasing towards the bottom.



2.3 Formations

The game will be structured using blocks called *formations*. A formation can represent any object in the game with a position, size, and texture. Textures determine how the formation looks when rendered in the UI. They are string codes like defined in the `Constants.TEXTURES`[1] dictionary of `lab.py`. Here are renderings of some of the textures we will use:





This shows one of the zookeepers, a rock, food, and an animal.

Some formations might be moving formations. Those may have both an `x`-directional and a `y`-directional velocity, in units of distance traversed per timestep.

Throughout the game, collision detection will be based on the overlap between these rectangular formations. One formation intersects (i.e. collides) with another if any part of it overlaps with the other. Shared edges and corners do not count as intersections.

Note that formations may leave the board during the game. We say a formation has left the board if its center is no longer on the board. Coordinates on edges and corners of the board (such as `(0,0)`) are still on the board.

2.4 Game State

A `Game` is made up of three parts: a collection of formations, the zoo's money balance, and a global clock. Every time the clock ticks, the state of the `Game` is updated by appropriately modifying the money balance and all formations. This makes it simple to describe the `Game` as a class with three methods:

- `__init__(self, game_info)` initializes the game. `game_info` is a dictionary which contains the following necessary information for generating the game grid:
  - `'width'`: The integer width of the game grid.
  - `'height'`: The integer height of the game grid.
  - `'rocks'`: The set of tuples of rock coordinates. The coordinates are of the centers of the rocks, which have dimensions `Constants.ROCK_WIDTH` and `Constants.ROCK_HEIGHT`.
  - `'path_corners'`: An ordered list of coordinate tuples. The first coordinate is the start of the path, the last coordinate is the end (both of which lie on edges of the gameboard), and the other coordinates are corners on the path. Recall that "the path" refers to the unique path that animals follow as they traverse the board.
  - `'money'`: The player's money balance, in dollars.
  - `'spawn_interval'`: The interval (in timesteps) for spawning animals to the game. If the first call to timestep has count `0` and the spawn interval is `I`, an animal is spawned at timesteps with count `0`, `I`, `2*I`, . . .
  - `'animal_speed'`: The magnitude of the speed at which the animals move along the path, in units of distance traversed per timestep.
  - `'num_allowed_unfed'`: The number of animals allowed to finish the path unfed without the player being defeated.
- `timestep(self, mouse)` advances the game state by one time step based on player input `mouse`. In other words, `timestep` is responsible for simulating the evolution of the world across one unit of time. In each timestep, **if the game is** `ongoing`, the following changes or updates will occur, **in the listed order**:
  1. Compute any changes in formation locations, then remove any off-board formations.
  2. Handle any food-animal collisions, and remove the fed animals and the eaten food.
  3. Throw new food if possible.
  4. Spawn a new animal from the path's start if needed.
  5. Handle `mouse` input, which is the integer tuple coordinate of a player's click, the string label of a particular zookeeper type, or `None`.
  6. Redeem one dollar per animal fed this timestep.
  7. Check for the losing condition.
- `render(self)` tells our UI the state of the game by returning a dictionary of information (described in a later section).

See the later sections of this write-up for more detail on the steps that we have only sketched so far.

3 Object-Oriented Approach

The three methods in the above section are needed for the UI to display the game and are the only methods directly called by the test cases. Nonetheless, your solution should make use of many additional functions and classes. (Our solution has about 40 methods!) Doing so will simplify Lab 10, in which you'll extend your game. Think carefully about your design by considering the relationship between items in the game.

4 Game Setup

4.1 Serialization and Deserialization

Before we dive into game details, we need to understand how the `Game`'s state is communicated with the UI. The code skeleton that we have provided you is fairly minimal in order to provide flexibility. Therefore, there needs to be some standardized way of digesting the game's state. This logic is handled via the `render` function. In short, `render` outputs the game's state in a way that can be reconstructed later. This process is known as serialization. Conversely, the process of taking this normalized value and breaking it down into meaningful information is known as *deserialization*.

Your first task will be twofold:

1. *Deserialize* `game_info` in `Game`'s `__init__` and store all useful properties.
2. *Serialize* the state of `Game` in `render`.

More specifically, the UI will expect `render` to return a dictionary with the following entries:

- `'formations'`: A list of dictionaries in any order, each one representing a formation. Each dictionary has the key/value pairs `'loc': (x, y)`, `'texture': texture`, and `'size': (width, height)`, where `(x, y)` is the center coordinate of the formation, `texture` is its texture, and `width` and `height` are its dimensions. Zookeeper formations have an additional key in their dictionaries, `'aim_dir'`, which has value `None` if the zookeeper has not yet been aimed, or a tuple `(aim_x, aim_y)` representing a **unit** vector pointing in the direction that the zookeeper is aiming.
- `'money'`: The amount of money, in dollars, the player has available to hire zookeepers.
- `'status'`: The current state of the game which can be `'ongoing'` or `'defeat'`. It is `'ongoing'` initially.
- `'num_allowed_remaining'`: The number of animals which are still allowed to exit the board before the game status is `'defeat'`. (This number is less than or equal to the initial cap of `game_info['num_allowed_unfed']`.)

5 Animals

An animal is spawned (generated) on the first timestep. The new animal's center aligns with the path's start coordinate center. Another animal should be spawned on every `game_info['spawn_interval']`th timestep thereafter. An animal makes its way along the path with speed `game_info['animal_speed']`.

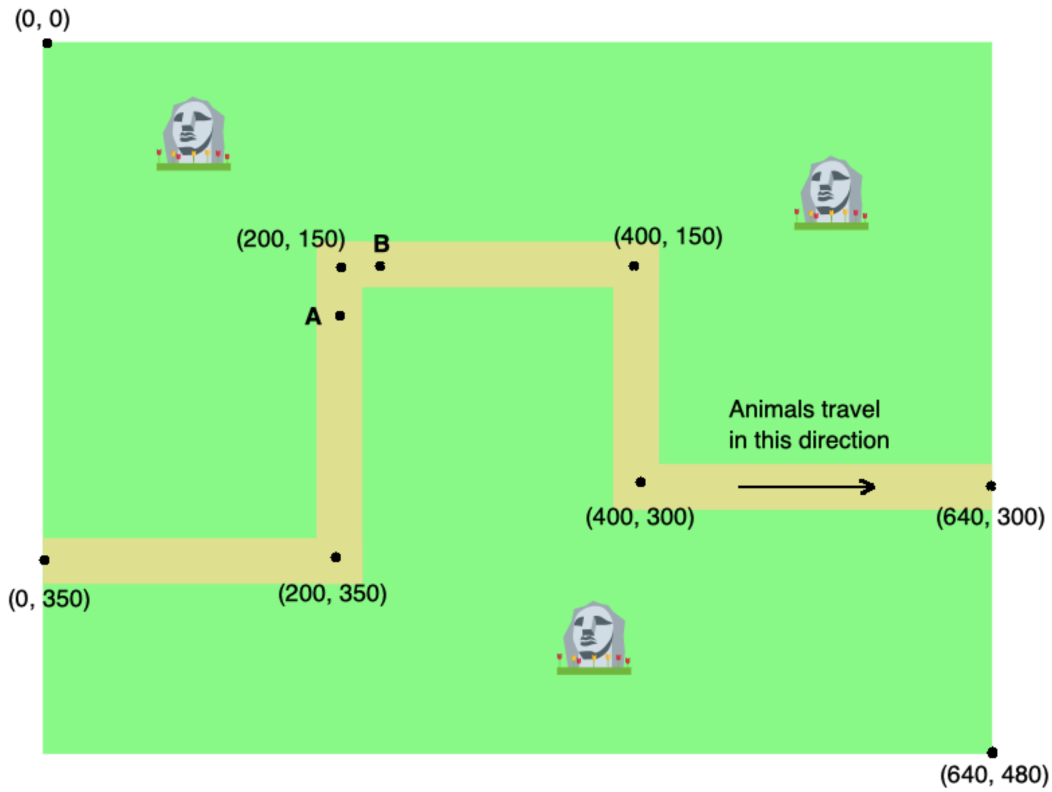
If an animal's *center* is no longer on the board or it is fed by one unit of food, it simply disappears from the game: your render should no longer return it. If **more than** `game_info['num_allowed_unfed']` animals leave the board, then the game status permanently becomes `'defeat'`, and the game comes to an end. Once this happens, the dictionary returned in `render` should no longer change, even after any number of calls to timesteps.

6 The Path

Each segment of the path will be parallel to one of the board edges. That is, every pair of consecutive coordinates in `game_info['path_corners']` share either their x- or y-coordinate. The coordinates in `game_info['path_corners']` all lie in the center of the path. Additionally, the path is guaranteed not to intersect any of the rocks. The path has a total thickness given by `Constants.PATH_THICKNESS`.

You're welcome to process `game_info['path_corners']` and store the path however you see fit.

Here's a board with just a path and some rocks, labeled with various coordinates and the direction of animal travel.



If `game_info['animal_speed']` is equal to the distance from **A** to `(200, 150)` plus the distance from `(200, 150)` to **B** , then an animal at **A** during one timestep will be at **B** in the next timestep. Animals do turn path corners, even within a single timestep. The distance animals travel is measured *along the path*.

7 Zookeepers

Zookeepers are how the player prevents animals from getting to the path's end without being fed! A Zookeeper is placed by the player at a permanent location on the board, and it can see animals in its line of sight. A future section will explain how zookeepers decide to throw food.

From your extensive research on zookeeper psychology and physiology, you know that there are exactly three types of zookeepers. They differ in their food throwing speed and frequency.

- `'SpeedyZookeeper'` has price 9, throw interval 55, and throw speed 20
- `'ThriftyZookeeper'` has price 7, throw interval 45, and throw speed 7

- `'CheeryZookeeper'` has price 10, throw interval 35, and throw speed 2

The zookeeper price is the amount of the one-time payment needed to hire the zookeeper; the throw interval is the number of timesteps during which the zookeeper makes at most one throw; and the throw speed is the speed at which the zookeeper's thrown food travels. The speed gives the total distance traveled per timestep of each unit of food thrown from the zookeeper, no matter the direction in which it is thrown. The `Constants` class in `lab.py` already defines each zookeeper's `price`, `interval`, and `throw_speed_mag` for you.

Here are their textures, shown within the UI's zookeeper choice panel:

7.1 Placement

To place a zookeeper, the user first clicks on the zookeeper type they want on the UI's zookeeper choice panel. The UI will interpret the location of this click and pass it into the `mouse` argument of the `timestep` function as one of the zookeeper labels (`'SpeedyZookeeper'` , `'ThriftyZookeeper'`, or `'CheeryZookeeper'`). Notice that your `Game` therefore need not know anything about the rendering or handling of the zookeeper choice panel – good abstraction!

After having chosen a zookeeper type, the user's next click indicates the center of the location at which they wish to place the zookeeper, and is passed into `mouse` as an integer tuple coordinate. Upon receiving this coordinate, you should **first** check whether the user can afford to pay the zookeeper, i.e. if the player's balance is greater than or equal to the price of the zookeeper. If it is not, raise a `NotEnoughMoneyError`, (defined for you in `lab.py`). If the player has enough money, you should **then** check whether the location is valid. A location is not valid if the zookeeper placed there would overlap with the path or any existing rocks or zookeepers. If you have enough money and the clicked location is valid, then the zookeeper is bought and placed there.

If the player has enough money to place the selected zookeeper, allow for multiple click attempts until the placement is valid. If the user would like to place multiple new zookeepers, they must reselect the zookeeper type from the zookeeper choice panel after every valid placement. This means that you should only "deselect" a keeper if keeper placement is successful.

Your implementation should deal with indecisive players: if a player clicks on multiple different zookeeper types before placing a zookeeper, the last-clicked type should be taken.

7.2 Aiming

Once a zookeeper is placed, the user's next click sets the aim direction of the keeper, the permanent direction in which that keeper will throw food. The click is passed into `mouse` as an integer tuple coordinate. **You should ignore the click if it is at the exact coordinate of that zookeeper.** Otherwise, you should set the aim direction of the keeper to be along the vector pointing from the zookeeper's center to the click. The zookeeper's line of sight extends from the zookeeper's center to the edge of the board, in this direction.

Notice that the magnitude of the vector from the zookeeper to the aim click therefore does not matter, and the zookeeper dictionaries returned by `render` require a *unit* vector `'aim_dir'` (one with magnitude 1).

You may assume that the user will aim a placed zookeeper before clicking on another zookeeper in the zookeeper choice panel.[2]

7.3 Throwing Food

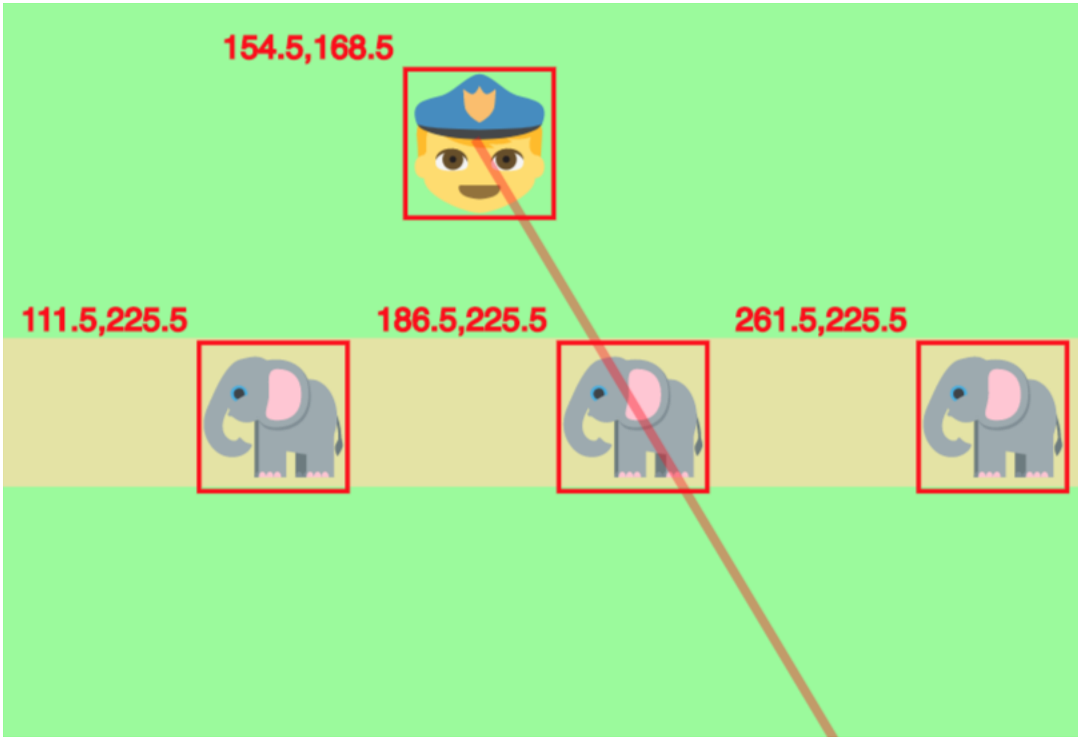
Once a zookeeper has been placed and aimed, they may begin helping the player feed animals. They do this by throwing food. A zookeeper may only throw food once during the number of timesteps which is its throw interval. More specifically, if the timestep *immediately after* that on which a zookeeper is placed has count `X` and the keeper's throw interval is `I`, then that zookeeper can only throw food on timesteps `X`, `X+I`, `X+2*I`, ... [3]

If the zookeeper can throw food on a particular timestep, they only do so if an animal falls in their line of sight (on that particular timestep). An animal falls in a zookeeper's line of sight if the line of sight intersects the animal formation (which has dimensions `Constants.ANIMAL_WIDTH` and `Constants.ANIMAL_HEIGHT`)[4].

Checking whether a line of sight intersects a rectangular animal is a nontrivial task. There are multiple approaches, and we encourage you to think about and explore them yourself, as a fun challenge. But we also describe one approach here:

First, some terminology: the zookeeper line of sight is a *ray*, a line which starts at an endpoint (the zookeeper's center), and extends off to infinity in one direction.

Notice that, as in the example image below, a ray intersects a rectangle if and only if it intersects at least one of the rectangle's sides[5]. Therefore, we can reduce the problem of checking for an intersection between a ray and a rectangle to the problem of checking for an intersection between that ray and any of four line segments.



Let the vectors v_{e1} and v_{e2} point from the endpoint of the ray toward an endpoint ($e1$ and $e2$, respectively) of the line segment. And let the vector v_{ray} give the direction of the ray. Then the line segment and ray intersect if

$\theta_{v_{e1},v_{e2}} = \theta_{v_{e1},v_{ray}} + \theta_{v_{ray},v_{e2}}$

where $\theta_{u,v}$ is the angle from v to u.

Note that since inverse trigonometric functions produce especially large floating point errors, in your code, you should check that these two values are equal *up to some small value*. Using a value of 0.001 suffices for our tests.

The angle between two vectors $\vec{u}$ and $\vec{v}$ is given by

$cos^{-1}(\frac{\vec{u}\cdot\vec{v}}{|\vec{u}||\vec{v}|})$

where $|\vec{v}|$ is the magnitude of $\vec{v}$, and $\vec{u} \cdot \vec{v}$ is the dot product of $\vec{u}$ and $\vec{v}$, which we have used before. To this end, we have imported the `acos` function (the cos^{-1} function) for you[6].

If food is thrown, it starts at the center point of the zookeeper that throws it. It is thrown in the zookeeper's aim direction, with the total magnitude of speed allowed by the zookeeper type. Food only reaches and feeds an animal if the food (a formation with dimensions `Constants.FOOD_WIDTH` and `Constants.FOOD_HEIGHT`) intersects the animal formation. **If the food intersects with an animal, then the animal is fed, and both the animal and the food disappear from the board. The player earns one dollar for feeding one animal. If the food does not hit an animal, it simply disappears when its *center* is no longer on the board.**

If multiple pieces of food intersect with an animal in a given timestep, remove all those pieces of food. If one piece of food intersects with multiple animals, each of those animals are fed. (Our animals are hungry, but happy to share!) Make sure to gain one dollar per animal fed, not per piece of food eaten.