Lab2: Artillery game

In this lab we extend the projectile example from the book into a small "artillery game" where two players take turns firing cannons, aiming to hit the cannon belonging to the other player.

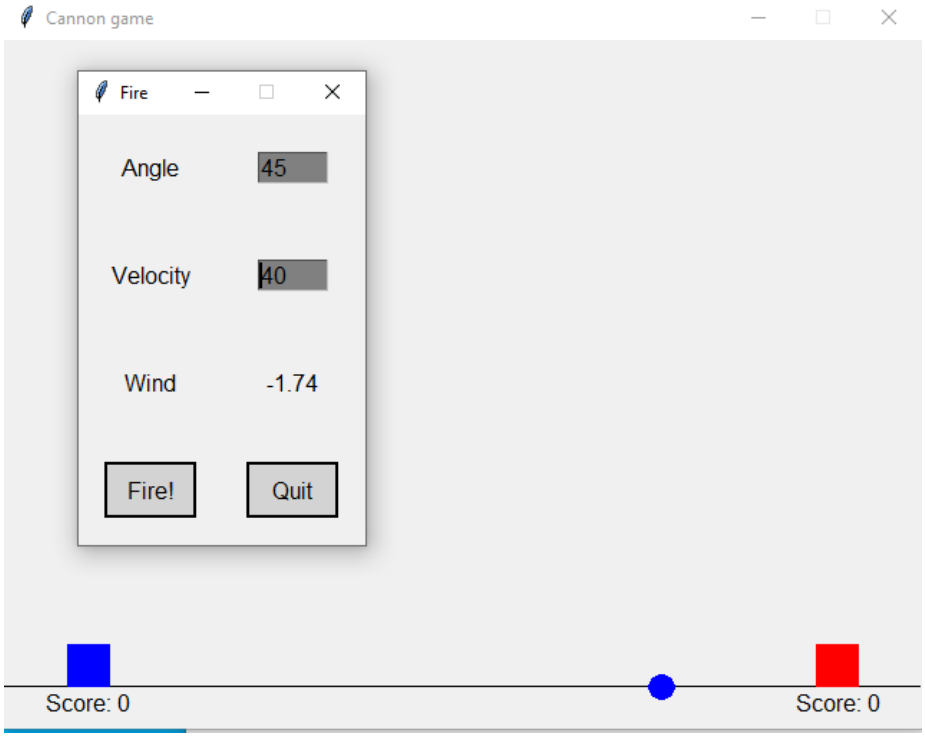


Figure 1: A screenshot of our game

(yes, the square blobs are cannons)

Here are the rules of the game:

- Round: A game consists of multiple rounds, and each round consists of multiple turns. A round ends when a player hits the other player's cannon, winning the round. When a round ends, the victorious player is given a point and the wind speed is changed to a new value. The starting player for each new round is the player who lost the last round.
- Turn: On each turn a player aims their cannons and fires, and then the turn passes to the other player. Note that wind speed does not change between turns, only between rounds.
- Wind: Wind speed starts as a random value between -10 and +10. When a round ends, a new random wind speed is generated. Wind speed affects the horizontal velocity of cannonballs. A negative wind speed accelerates can-

nonballs towards the west/left and a positive value towards the east/right. A high wind speed (near +/-10) affects the cannonball in the horizontal direction the same way gravity affects it in the vertical, so a cannonball could change direction and "fall" back towards the player that fired it! The unit for wind speed is m/s2, a few observations should be made about this:

- It is not actually the speed of the wind, rather it is the constant acceleration that enacts on all cannonballs. This is not an accurate model of how wind works in real life, but it is close enough to make it feel realistic and to make the game interesting.
- The fact that it uses meters and seconds has few implications on the game, internally it uses abstract units of distance and time and the simulation can be sped up or slowed down. However the way we implement gravity for cannonballs implies that the units are meters and seconds.

Introduction

With this lab, we learn to structure code using classes and objects. We also learn some basic graphics using the graphics.py module provided with the course book. A very important aspect of this lab is the separation of the model (the code that deals with game rules, and the logic of playing the game) from the graphics/user interface code that deals with displaying cannons and cannon balls and pressing buttons to make things happen. The idea is that the graphics code uses the model a lot (asking it for information and telling it to perform actions) but the model never directly deals with graphics. To show that the model works independently of user interface we will equip our game with two different user interfaces that work against the same model: A textual interface used for testing and a graphical user interface that is more useful for actually playing.

Examples of what belongs in the model: - Representing the players and determining whose turn it is, and the current score of each player. - How a projectile moves as simulation time progresses (wind speed, gravity etc.). - Determining if a projectile hits or misses its target.

Examples of what clearly belongs in the graphics part: - User input (buttons, text fields etc). - Everything that deals with the graphics.py module directly.

Examples of things that are less obvious: - Each player is identified by a color, this is a string and is part of the model but it needs to be a color recognized by the graphics library. - The sizes of projectiles and cannons are part of the model since they are needed to determine if a projectile hits a cannon, but they are also used to draw the cannons and cannon balls.

You are given a fair amount of code from the start, mainly to keep the workload reasonable but also for you to practice working on, understanding and extending code written by someone else.

Tasks

This assignment is divided into three parts: - Part 1: First we implement and test the game model. - Part 2: Then we add graphics to our game and test it to some degree - Part 3: Finally we construct the main game loop to make the game playable. If you prefer, you can interleave these three and alternate between developing the model, the graphics and the game loop.

Setup

Download the archive lab3.zip which contains all the files needed for the assignment. These are: - gamemodel.py: Contains the model of the game, modify it in Part 1. - gamegraphics.py: Contains the graphics of the game, modify it in Part 2. - main.py: This file runs the game with the graphical interface, modify it in Part 3. - test.py: Runs tests on your solution, do not modify. - textmain.py: Runs the game using a textual interface, no need to modify. - graphics.py: The graphics library provided by the course book, do not modify. Documentation is found here: https://mcsp.wartburg.edu/zelle/python/graphics/graphics.pdf

Note that although you should not modify test.py, studying it will be very helpful in completing the lab. For example, lines 15-17 in that module are:

```
players = game.getPlayers()

test(len(players) == 2, "there should be two players")
```

From this we see that method getPlayers() must return a list with two elements, e.g. two players. If you haven't done anything with the lab, yet, that test will fail since the skeletal implementation that you got returns an empty list:

```
    """ A list containing both players """
    def getPlayers(self):
        return [] #TODO: this is just a dummy value
```

Solving that problem can be the first step in your solution. When you have passed all the tests, you will be done with the assignment.

Part 1: Constructing and testing the game model

For this part we will be working with gamemodel.py, the module that contains all our model classes. We will also be running the tests from test.py as we go. We have three model classes:

- `Game`: The top-level class of the model. A `Game` object indirectly contains the whole logical state of an ongoing game. It directly contains global data like wind speed and whose turn it currently is. The sizes of cannons and cannonballs are also stored directly in the `Game` class. Methods in the class are largely for accessing this data and commands for starting new rounds and such.

- `Player`: Each `Player` object represents a player. A `Game` object contains two `Player` objects, although in principle the game can be extended to add more players. `Player` objects contain data like current score, the position of the player's cannon and the current angle and velocity settings of the player's cannon. Each player also has a direction of fire, for the same angle value the player to the left should fire in the opposite horizontal direction of the player to the right.
- `Projectile`: A `Projectile` object represents a cannonball that is either moving or resting after hitting the ground. Each `Player` object has either a single `Projectile`, or no projectile if the player has not yet fired. Each time a player fires its cannon; a new `Projectile` is created and replaces the previous one. The primary attributes of a projectile are position and velocity, and it has a method to advance simulation time by a given amount and adjust values based on simplified laws of physics. Compared to `Game` and `Player`, `Projectile` is a more general class, usable to model any kind of projectile and is dependable on very few game-specific concepts. This class is given and you do not need to modify it.

In the package for this assignment the file gamemodel.py already contains class definitions and method definitions, but the actual bodies of the methods are just placeholders, either returning a nonsense value or using "pass" to do nothing. The type of the "dummy" return values reflects the type the methods should return: numbers, strings, Booleans etc. "None" is used for values that should return objects, and pass is used for methods that return nothing.

The following diagram shows how the objects of the different classes should look like when they are instantiated in the program:

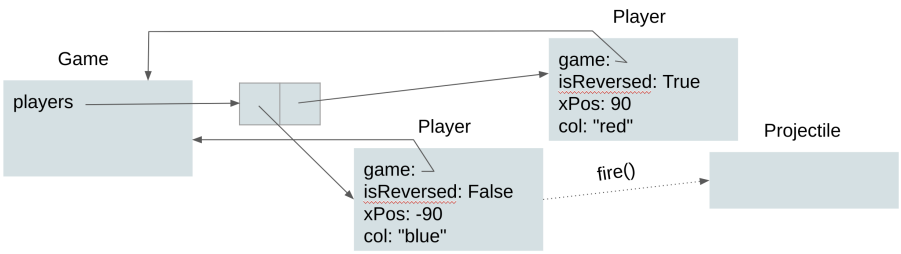


Figure 2: The data model of our game

`Game` for instance should contain an attribute which points to a list of players. Each player (an object of class `Player`) on the other hand should have an attribute which points back to the game to which it belongs. This allows methods in `Player` to access information that is stored in the `Game`. Both `Game` and `Player` have other attributes as well. For instance on the diagram we showed the attributes for color and position. The attribute isReversed accounts for the fact that the first player must shoot from left to right, while the second one shoots from right to left. For compactness, we did not show all attributes. You

will find that more are needed while reading further.

The `Projectile` object is created and must be returned by the `fire()` method. That is why on the diagram we used a dashed arrow. The `Player` object does not have to save a reference to the `Projectile` as an attribute. It is enough to create it and return it to be used by the rest of the program.

A good place to start this part of the assignment is the `__init__` method for `Game`. It should create the two players and add them in a list which on the other hand must be stored as an attribute to the object. When creating the players, they should be initialized with the right position, color and shooting direction. As discussed before, the player must also have access to the game object to which it belongs. To make all that possible, you should define an appropriate `__init__` method for `Player`, which initializes all of its attributes. After that, here is an example for how the players can be created from within the `__init__` method for `Game`, if you have already done the `__init__` method for `Player`:

```
[Player(self, False, -90, "blue"), Player(self, True, 90, "red")]
```

Here `self` will refer to the game itself, if the line is part of its `__init__` method. Finishing everything in that paragraph will get you through the first case in test.py.

For the next four test cases, you need to finish methods `getColor()` and `getX()` in `Player`. Their implementation is trivial given that you have implemented properly the `__init__` methods for `Game` and `Player`.

Finnishing methods `getCurrentPlayerNumber()`, `getCurrentPlayer()`, `getOtherPlayer()` and `nextPlayer()` will resolve the next 10 test cases. At any point we should keep track of whose turn it is to play. You can do that by saving as an attribute the index of the current player. We initialize the index to 0 in the `__init__` method and we flip it from 0 to 1 and vice versa each time when `nextPlayer()` is called. Given that, `getCurrentPlayerNumber()` must simply return the current index, while `getCurrentPlayer()` returns from the list of players the one with the right index. Method `getOtherPlayer()` must return the other player, i.e. if the current index is 0, then it returns the player with index 1 and vice versa.

The next methods to implement are `getCurrentWind()` and `setCurrentWind()`. They must simply return/modify an attribute in `Game` which keeps track of the wind speed. According to the rules of the game, the game starts with a random wind speed in the range -10 to 10. The speed also changes to a new random value at each new round. This means that both `__init__` and `newRound` must set a random value to the corresponding attribute. In Python, you generate a random number between 0 and 1 by using `random.random()`.

By now all methods in `Game` are implemented, except `getCannonSize()` and `getBallSize()`. Note that when the game is created its `__init__` method receives both the cannon size and ball size as arguments. Those must be save in attributes and returned by `getCannonSize()` and `getBallSize()`.

The next step is to make it possible to fire cannon balls. In other words, we need to implement the method `fire()` which creates and returns a new projectile. The `__init__` method for `Projectile` looks like this:

```
def __init__(self, angle, velocity, wind, xPos, yPos, xLower, xUpper):
    ...
```

It takes an angle and velocity which you get as arguments to `fire()`. Note, however, that the second player must shoot in reverse direction. This means that when `isReverse==True`, instead of passing `angle` to the projectile, you should pass `180-angle`. The wind you can get by calling `getCurrentWindow()` from `Game`. This of course will be possible only if you have saved a reference to the game in every player. `xPos` and `yPos` are the initial coordinates of the cannon ball. The cannon ball is always launched from the center of the square blob which represents the cannon. This means that the initial x coordinate for the cannon is the same as the x coordinate for the player. The y coordinate on the other hand must be half the size of the cannon. The cannon size you can again get from the game, if you have implemented the method `getCannonSize()`. Finally, the last two arguments that you need to create a `Projectile` are `xLower` and `xUpper`. They describe how far to the left/right the projectile can go. You can you use the hard-coded values -110 and 110 for these. The motivation for all the coordinates is also shown on the image bellow:

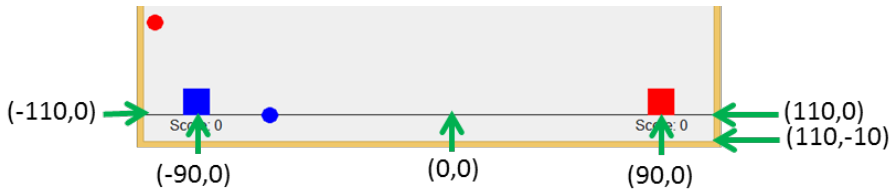


Figure 3: Positions

Once we can fire cannon balls, we must also be able to detect whether the other player is hit. The method `projectileDistance()` makes that possible. The idea is that `p.projectileDistance(proj)` answers the question "by how much did the projectile `proj` miss the cannon of player `p`?". If the cannon and projectile are touching it should always return 0. If they are not touching it should give the shortest distance from the edge of the cannonball to the edge of the cannon. Graphically, the green dashed line in this image shows the projectile distance from the blue player to the red projectile.

The x-positions of the cannon and cannonball are both given as the center, so you need to compensate for the width of the cannon and the radius of the cannonball.

The last method involved in firing and hitting is `getAim()` which must return a tuple of the last angle and velocity used by the player for firing. To make that

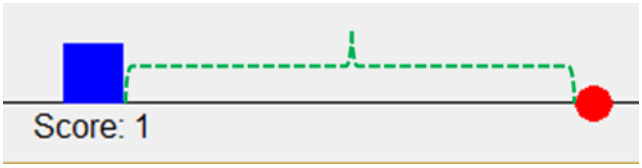


Figure 4: Distance

possible, just save the angle and velocity in attributes, every time when `fire()` is executed.

At this point we are almost done with Part 1. What is left is methods `getScore()`, `increaseScore()`, `getColor()` and `getX()`. For the first two you need an attribute which is first initialized to zero in `__init__`, and then incremented every time when `increaseScore()` is called. The last two should return the color and the x coordinate, which you can save as attributes in the constructor.

Now we finished Part 1 of the assignment, and we can play the game using a crude textual interface provided for you in textmain.py.

Part 2: Adding Graphics

The game graphics is based on three classes that mirror the classes in the game model, extending each with additional functionality related to graphics. - `GraphicGame`: Wraps around a `Game` object. Provides everything `Game` does and deals with creating the main graphical window and the "terrain" (just a flat line in this game). - `GraphicPlayer`: Wraps around a `Player` object. Provides everything `Player` does and deals with drawing the players cannon and displaying the player's score correctly. - `GraphicProjectile`: Wraps around a `Projectile` object. Provides everything `Projectile` does and deals with drawing the cannonball as it hurls across the sky.

Additionally, the gamegraphics.py have two classes for graphical components that have no corresponding classes in the model: - `InputDialog`: shows a separate window where users can adjust their aim and either fire or quit the game. This class is given and you should not need to modify it. - `Button`: a general class for creating buttons, taken from the course book. This class is given and you should not need to modify it.

Now what does "wrap around" mean? The diagram bellow shows how the entire state of the program will look like when graphics is added:

We have two layers: computation and visualization. Part 1 of the assignment was all about the computational layer. The graphical layer consists of three similar classes, e.g. `GraphicsGame`, `GraphicsPlayer`, `GraphicsProjectile`. Each of them contains a reference to its computational counterpart, e.g. on the diagram, these are the attributes `game`, `player` and `proj`. If you look at the existing code in gamegraphics.py, you will see that most of the methods for `GraphicProjectile`

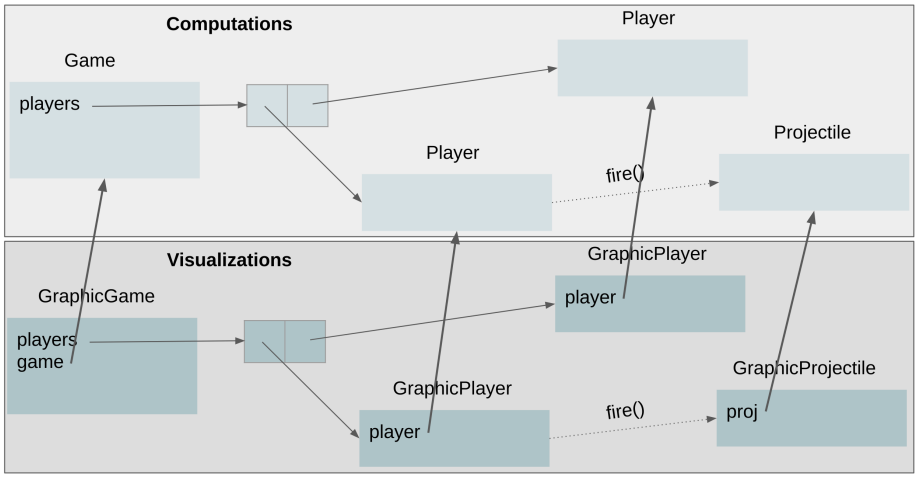


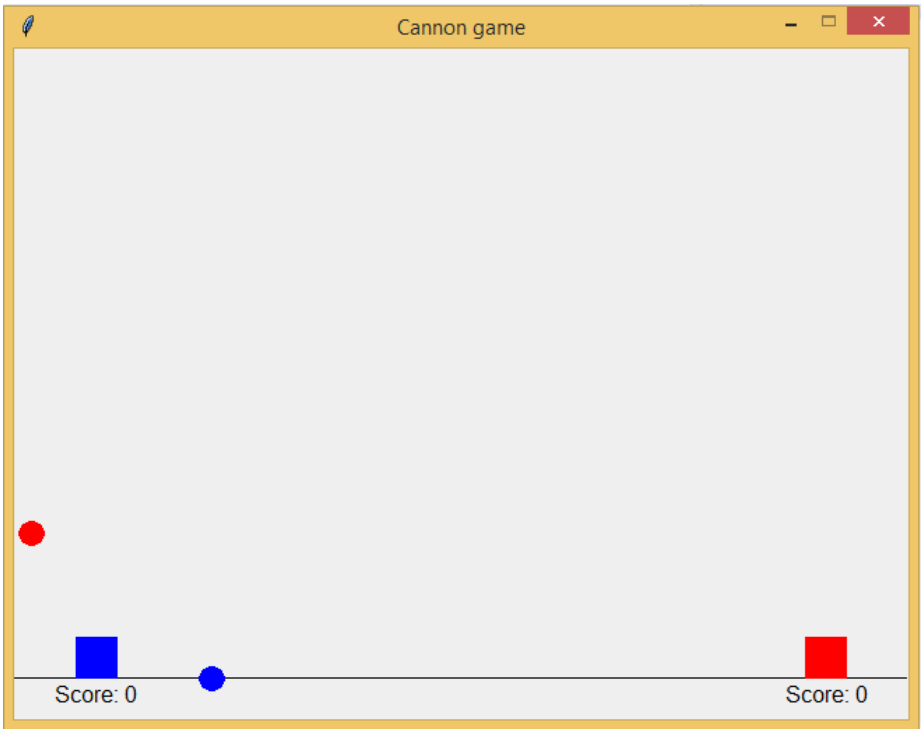
Figure 5: The data model together with the graphics

and `GraphicPlayer` are already implemented by just calling the corresponding methods from `Projectile` and `Player`. What is missing is to add `__init__` methods which initialize the attributes `player` and `proj`.

Something similar needs to be done for `GraphicGame` too. Note, however, that while `Game` creates two `Player` objects, `GraphicGame` must create two `GraphicPlayer` objects and therefore it has its own attribute `players`. Any method that returns a Player or Projectile should instead return `GraphicalPlayer`/`GraphicalProjectile` objects. E.g. `getCurrentPlayer()` in `GraphicGame` should give a `GraphicPlayer`, and if you accidentally return a `Player` then graphics will not work. Similarly fire() must return `GraphicalProjectile` or otherwise you won't see anything happening on the screen.

In programming terminology, what we did is called "wrapping". We created new classes which delegate most of their work to other classes, but they also add something on their own. In our case the new classes must add graphical visualization. Note that you should not copy any of the code from the base classes! `GameGraphics` should work by calling methods in `Game`, not doing any work of its own.

A good programming strategy is to delegate to each class only responsibilities that it can fulfill. The separation of responsibilities should be such that in future developments, it should be intuitive to find which parts of the code need changes and every change should be as local as possible. This improves maintainability and reduces the cost of software development. In our case we have three classes and we need to decide which class draws what. This figure shows all the graphical elements:



It is obvious that only the `GraphicProjectile` knows the current coordinate of a cannon ball, and therefore it should be responsible for drawing the circles. Similarly it is natural that `GraphicPlayer` should draw the cannons and the current scores, while `GraphicGame` takes care of the global elements, i.e. it should create the window and draw the terrain. For convenience, we have marked where the graphical visualization should happen as TODO comments in the code. This is also summarized bellow:

- `__init__` in `GraphicGame` must open a window and draw the terrain. *Hint:* This code creates a graphical window and modifies its coordinate system to match that of the game, so coordinates in the model correspond to coordinates in the graphical window: `Python win = GraphWin("Cannon game" , 640, 480, autoflush=False) win.setCoords(-110, -10, 110, 155)` The height of 155 is chosen to match the proportions of the game window (640x480 actual pixels vs. 220x165 logical pixels)
- `__init__` in `GraphicPlayer` must draw the cannon and the current points for the player.
- `fire()` must undraw the old projectile for the player. In order to do that each time when fire() is called you must save the newly created projectile in an attribute, in order to have access to it from the next call to the method.
- `increaseScore()`, must update the text for the current scores after each increment. *Hint:* You should have created and drawn a `Text` object in the

`__init__` method for the player. In the beginning that object would just draw the initial score "0". Save a reference to the object in an attribute, and each time when `increaseScore()` is called, update the text by calling `setText()` from the text object.

- `__init__` in `GraphicProjectile` must create a `Circle` and draw it. Save a reference to the circle object in an attribute.
- `update()` in `GraphicProjectile` must update the visualization. Use `move(dx,dy)` from the `Circle` class in graphics.py
- In `GraphicProjectile` add a new method `undraw()` which must undraw the cannon ball. The method should then be used in `fire()` to undraw the last cannon ball before a new one is launched.

Testing the game graphics

After test.py is done testing the model, it will proceed to run the same tests on the graphical game. This has two benefits: 1. It ensures our graphical classes give indistinguishable results from their corresponding model classes. 2. It will create a main window that should contain several graphical component if our classes work as intended.

It then proceeds to run some additional graphics-related tests, like counting how many circles are actually visible in the window and such.

This is what the window created by running test.py should look like when gamegraphics.py works as intended:

Note that the score has been updated, and there are exactly two visible cannonballs.

Note that the window is created instantly without time passing. Delaying the simulation is handled by the main loop.

Part 3: Implementing the game loop

Now that we are reasonably sure we have a correct game model and graphics, it is time to create the main loop of the game in main.py. It may be a good idea to have the same separation of each turn into phases as in the textual interface (check textmain.py): - *Input phase*: Use the `InputDialog` class to request user input (angle and velocity) - *Firing phase*: Use the provided `graphicFire()`-function provided in main.py to fire and animate a cannonball. - *Cleanup phase*: Award points and start a new round if the projectile hits. Cycle to the next player (regardless of hit or miss).

Note that your game loop will be a lot "cleaner" than the one for the text based interface that does a lot of printing and stuff, because all the graphics are already handled by the graphics classes. For instance when you call `p.increaseScore()` and p is a `GraphicPlayer` you don't have to do anything extra to show the updated score, that is handled automatically by the class.

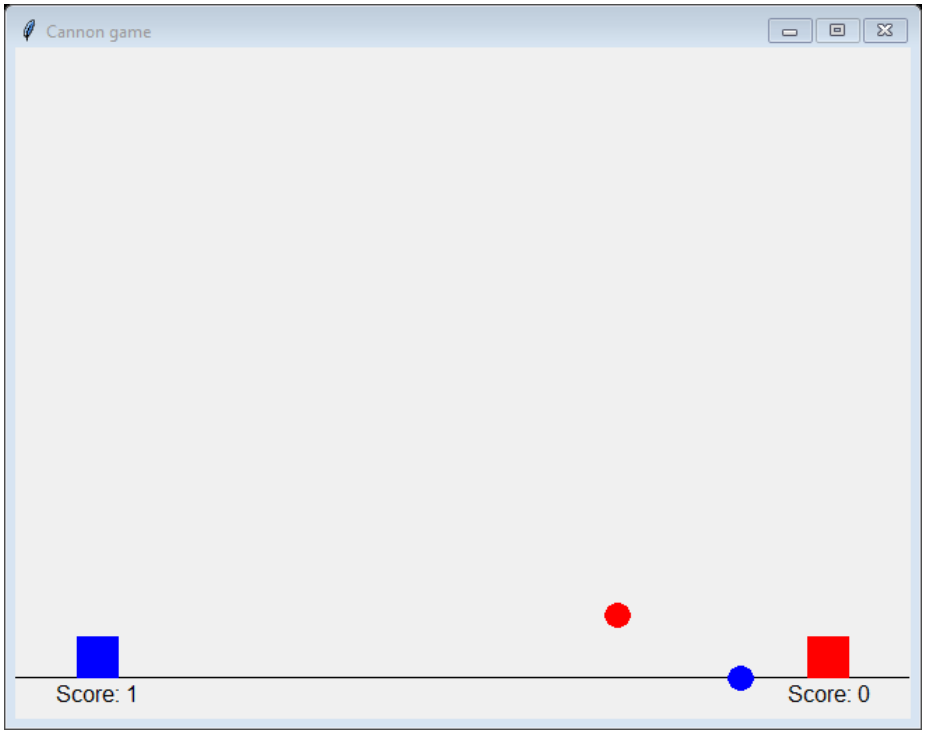


Figure 6: Screenshot 3

Submitting your solution

Upload the files gamemodel.py, gamegraphics.py and main.py to Canvas