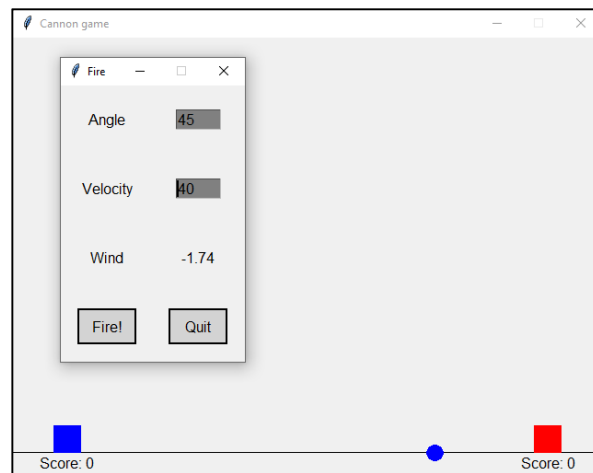


## Lab2: Cannons

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

### TL;DR

For those who have no patience for long explanatory texts: Your assignment is to finish all methods stubs and TODOs in the classes of `gamemodel.py` and `gamegraphics.py` until they pass all assertions run by `testgame.py`, then add a game loop in `main.py` to make the game playable with a graphical interface.



A screenshot of our game  
(yes, the square blobs are cannons)

## Introduction

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 (code that deals with game rules, and the logic of playing the game) and 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 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 them 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 all the files from the assignment page. These are:

- main.py: This file runs the game with the graphical interface, modify it in Part 3.
- 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.
- testgame.py: Runs tests on your solution. You should not modify this file but studying it can be a big help to know what you should be doing.
- 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 found here:

<https://mcsp.wartburg.edu/zelle/python/graphics/graphics.pdf>

## Design

The high-level design of your solution will largely be given, but this section will explain the reasoning behind the chosen design and some alternative designs that would be possible.

The design of the model is centered on a handful of classes in a single module (gamemodel.py):

- **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 players cannon and the current angle and velocity settings of the players cannon. A player can also be reversed, meaning it fires in the opposite horizontal direction compared to a non-reversed player even if they are given the same angle and velocity values.
- **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 referencing very few game-specific concepts. This class is given and you do not need to modify it.

The game graphics are 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 players 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 game graphics 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.

## Game loop

The game is played in rounds, consisting of multiple turns. A round ends when a player hits the other players cannon. The main game loop runs until the game exits, and each iteration

is a single turn. The actions the loop performs each turn can be described in this sequence of tasks:

- Request input (velocity and angle)
- Tell a Player object to fire a projectile
- Animate the projectile until it stops
- Check hit or miss. On a hit, award the Player a point and tell the Game to start a new round.
- Tell the game to switch to the next player.
- Repeat

## Design choice: Level of encapsulation

One possible design for the model is to just have a single method for firing the cannon of the current player with given angle and velocity, simulating its whole trajectory, calculating hit or miss and awarding points. This solution would have a high level of encapsulation, from an outside perspective there is no risk of calling methods in incorrect order or such, the game loop would only consist of requesting input and calling this method repeatedly.

The problem with this design is that there is no easy way to hook our graphics into the model. In order to build our graphics separately from the model we need to expose a richer interface to the user of our model. This means we will allow for instance awarding points to players even if they have missed and similar things that break the game rules.

## Design choice: Inheritance or delegation?

It seems like implementation inheritance is perfect for the graphic classes, i.e. GraphicPlayer extends Player etc. However, this would cause problems e.g. if we want the Game class to create its own Player objects. If GraphicGame extends Game, and Game creates its own Player objects there is no simple way to turn those Players into GraphicPlayers. There are various ways of working around this, e.g. by parameterizing Game over the kind of players it creates, but we want to keep things simple.

Instead we use specification inheritance and delegation to hook our graphics into our game model. The idea can be described as follows (for the Player class):

- Each GraphicPlayer object contains a Player object, "wrapping" it in graphic.
- GraphicPlayer has all the same method names as Player, with the same parameters (and possibly some additional methods not in Player).
- Most methods in GraphicPlayer simply delegate to its underlying Player object, e.g. if we call `p.getScore()` on a GraphicPlayer `p`, it will just return the result of running `getScore()` on its underlying Player object.

Taken together, this means that from the outside a GraphicPlayer is almost indistinguishable from a regular Player. Although it is possible to determine that a GraphicPlayer object is not a Player object, they work similarly enough for our purposes. This pragmatic approach to specification inheritance is called duck typing ("If it walks like a duck and it quacks like a duck, then it must be a duck") and it is commonly used in Python.

A downside of this approach is that it's easy to add a method to `Player` and forget to add the corresponding method in `GraphicPlayer`. We mitigate this risk by running the same tests cases on our model classes as on our graphical classes, and hopefully that means they are similar enough for our application to work.

## Game concepts and rules

This section explains some details of the game in a non-technical way and introduces some concepts later used in the assignment.

**Round:** A game consists of multiple rounds, and each round consists of multiple turns. A round ends when a player hits the other players' 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:** 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 value is generated. Wind affects the horizontal velocity of cannonballs. A negative wind value accelerates cannonballs towards the west/left and a positive value towards the east/right. A strong wind (near +/-10) affects the projectile 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  $\text{m/s}^2$ , a few observations should be made about this:

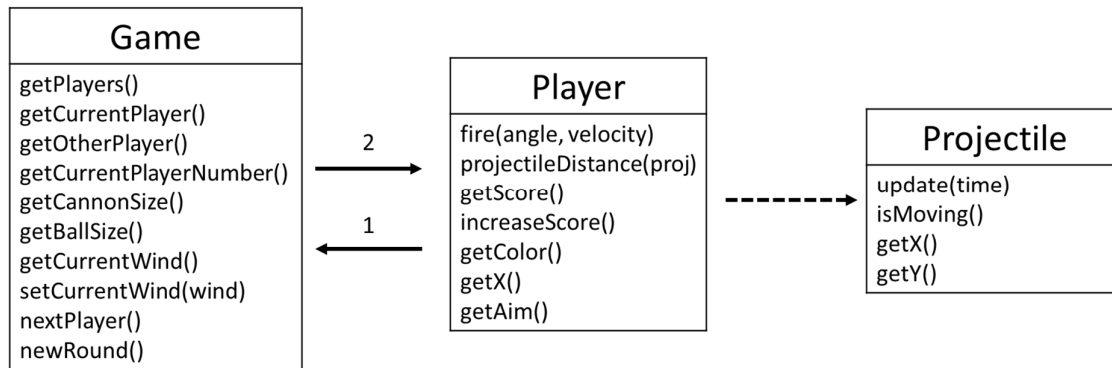
- It is not actually the speed of the wind, rather it is the constant acceleration it enacts on all cannonballs. This is not an accurate model of how wind works in real life, but it is close enough to make it feel realistic and to make the game interesting.
- The fact that it uses meters and seconds 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.

## 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 `testgame.py` as we go.

On the assignment page you will find a “skeleton” for `gamemodel.py`, it 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.

In the notation used in the book, our classes are connected like this:



Implementing all these methods may seem like a lot of work, but the majority of these are simple one-liners and you are given some code to get you started, including the entire Projectile class.

The notation on the Game-Player arrows means a Game contains two Players, and each Player in turn contains the Game that spawned them. Of course, “contains” here just means having a reference to, so two objects “containing” each other is perfectly possible.

The dashed arrow means player creates projectiles (but does not need to contain them).

## Learning from the test file (testgame.py)

**Note:** You should not modify testgame.py!

There are comments in the “skeleton” of gamemodel.py to specify what methods should do, but in some ways the test file is a more precise specification and looking at it may answer a lot of questions about the intentions of the various methods.

For instance, let us look at the first couple of lines from the tests:

```
def runTests(game):
    players = game.getPlayers()

    assert len(players)==2, "There should be two players!"

    # Testing game initialization
    assert players[0].getColor()=="blue", "player 0 should be blue!"
    assert players[1].getColor()=="red", "player 1 should be red!"
```

The assert command stops the execution if the condition is false. The second parameter of each assertion is an error message to be displayed in case it fails. Passing assertions have no effect, so for a fully working model, runTests should have no visible effect.

The tests are executed using `runTests(Game(10, 3))`, so the game parameter is a freshly created Game object. The parameters specify that cannons are 10 meters wide and cannonballs have a radius of 3 meters (this affects what is considered a hit). The first assertion tells us that calling `getPlayers()` on our freshly created Game should give us a list with two items (naturally these should be Player objects), and so we know that the constructor for Game has to create two players. The second and third assertions tells us that the color property of the first player should be “blue”, and for the second “red”.

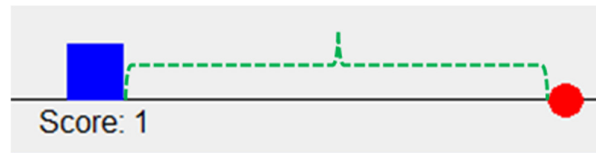
Modifying the classes to pass these assertions should be simple, and then re-running the tests should cause a brand-new assertion to fail, giving you a natural next step to work on.

Note that the tests are not perfect, for instance having two references to the same player in the list and having that player alternate between answering blue and red for its color would pass, but that is clearly not what the specification intends. This means that even if your model passes all assertions, it may still not work as intended.

## Game and Player classes:

Look at the comments of each methods to get some hints. Here are a few things to consider:

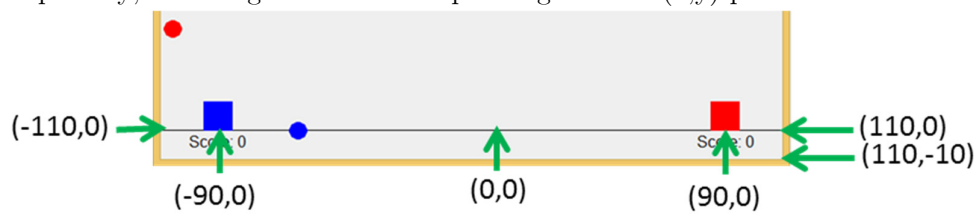
- The methods `getPlayers()`, `getCurrentPlayer()`, `getOtherPlayer()` and `getCurrentPlayerNumber()` in `Game` are all closely related and provide two different ways of determining who is the current player. Make sure that your internal representation only stores this information once (having a variable for each of these getters would be bad design, easily causing bugs!).
- Each `Game` objects naturally refer to its players, but it is convenient for the players to also know which game they are part of. Since the game creates the players, it will have to pass itself to the constructor of the `Player` class.
- The `projectileDistance()` method in `Player` can be a bit tricky. 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 give 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 measure 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

- Regarding positions: In the graphical interface, coordinates are never visible to the player, but they are used in the textual interface and internally for various computations. The ground is Y-position 0, increasing upwards. X-positions range from -110 to +110 from left to right and the cannons are placed at -90 and +90, so the point (0,0) is on the ground exactly between the cannons.

Graphically, this image shows arrows pointing at some (x,y)-positions.



## The Projectile class

The Projectile class is largely based on the Projectile class from the book, and you are given a working class. You may need to study the class a bit to understand how to use it, but you should not need to modify the class. The constructor has a lot of parameters (wind speed, upper and lower bounds for x-position ...).

The update() method takes a time value in seconds, and moves the projectile based on its current speed and acceleration (from gravity and wind).

## Interlude: Playing by text

After completing the model of the game, it is possible to play the game using a crude textual interface provided for you (in textmain.py).



## Part 2: Adding Graphics

A good start for this part is setting up the `GraphicGame`, `GraphicPlayer` and `GraphicProjectile` classes to work exactly as their model counterpart. This is already done for `GraphicProjectile` and more or less for `GraphicPlayer`. The `GraphicGame` class is not created at all so you will need to create it from scratch.

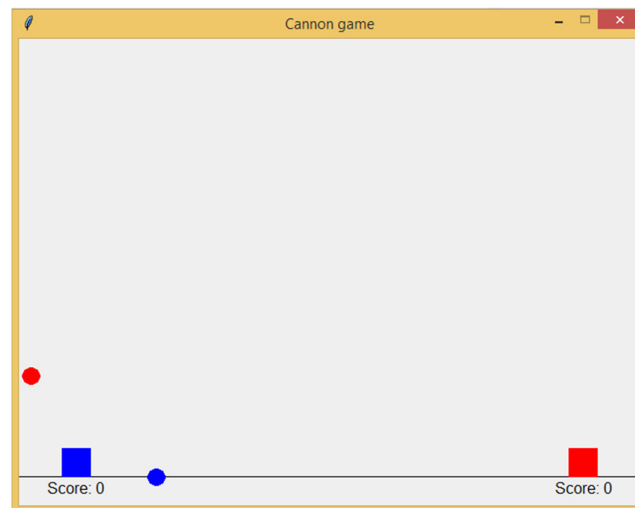
Note that you should not copy any of the code from the base classes! `GameGraphics` should work by calling methods in `Game`, not doing any work of its own.

After our `Graphic*`-classes work exactly as their base counterparts, we modify some of the methods to draw all the graphical components and update them when needed. Refer to the documentation of `graphics.py` (or the source code if you feel brave) to learn how to create and manipulate graphical components.

One important change: Any method that returns a `Player` or `Projectile` should instead return `GraphicalPlayer`/`GraphicalProjectile` objects.

### Separating responsibilities

Each visible graphical element is the responsibility of one of the classes.



`GraphicGame` creates the main game window and draws the line that represents the ground.

The `GraphicPlayer` object for the blue player draws the blue cannon and the score for the blue player (and the `GraphicPlayer` for the red player does the same for red). The `GraphicPlayer` objects also deal with creating and removing the `GraphicProjectiles`. At any point there should be at most two visible projectiles, as soon as a cannon fires a new projectile its old projectile should be removed.

The `GraphicProjectile` objects deal with drawing the cannonballs (the circles) and moving them as time progresses. All the computation of where it moves to is handled by the base `Projectile` class, `GraphicProjectile` just needs to make sure the move is reflected in the game window.

The Separate input window and its buttons are handled by the InputDialog and Button classes. These are given to you and you do not need to modify them.

For GraphicGame:

- Create the main graphical window in the constructor.  
**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:  

```
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 (640\*480 actual pixels vs. 220\*165 logical pixels)
- Create two graphical players in the constructor, each wrapping around one of the players in the underlying Game object.
- Modify getCurrentPlayer, getOtherPlayer, and getPlayers to return GraphicPlayers instead of Players. Note that it should not create new GraphicPlayers, just return already existing ones.

For GraphicPlayer:

- Make the constructor draw the players cannon (a square centered at the x-position of the player with its lowest part touching the ground) in the main window. The Game object knows the size of cannons, and the Player object knows the color and position of this specific cannon.
- Draw a Text object in the main window containing this players score (just below the cannon)
- Modify fire() to do two things:
  - 1: If the player already has a GraphicProjectile from a previous turn, undraw it (you need to add a method to GraphicProjectile for undrawing).
  - 2: Wrap the projectile you get from the fire()-method in Player in a GraphicProjectile, store it away (so it can be undrawn the next time you fire) and return it.
- Whenever the score of a player is increased you need to update the score text field to reflect the current score of the player.

For GraphicProjectile:

- The constructor needs to create and draw a circle. It needs access to the GraphWin object (the game window), the radius of cannonballs and the color of this specific cannonball in order to do this.
- The update method needs to move the circle after the underlying projectile has been updated, to reflect the new position of the projectile.
- Add an undraw() method to remove the projectile from the game window.

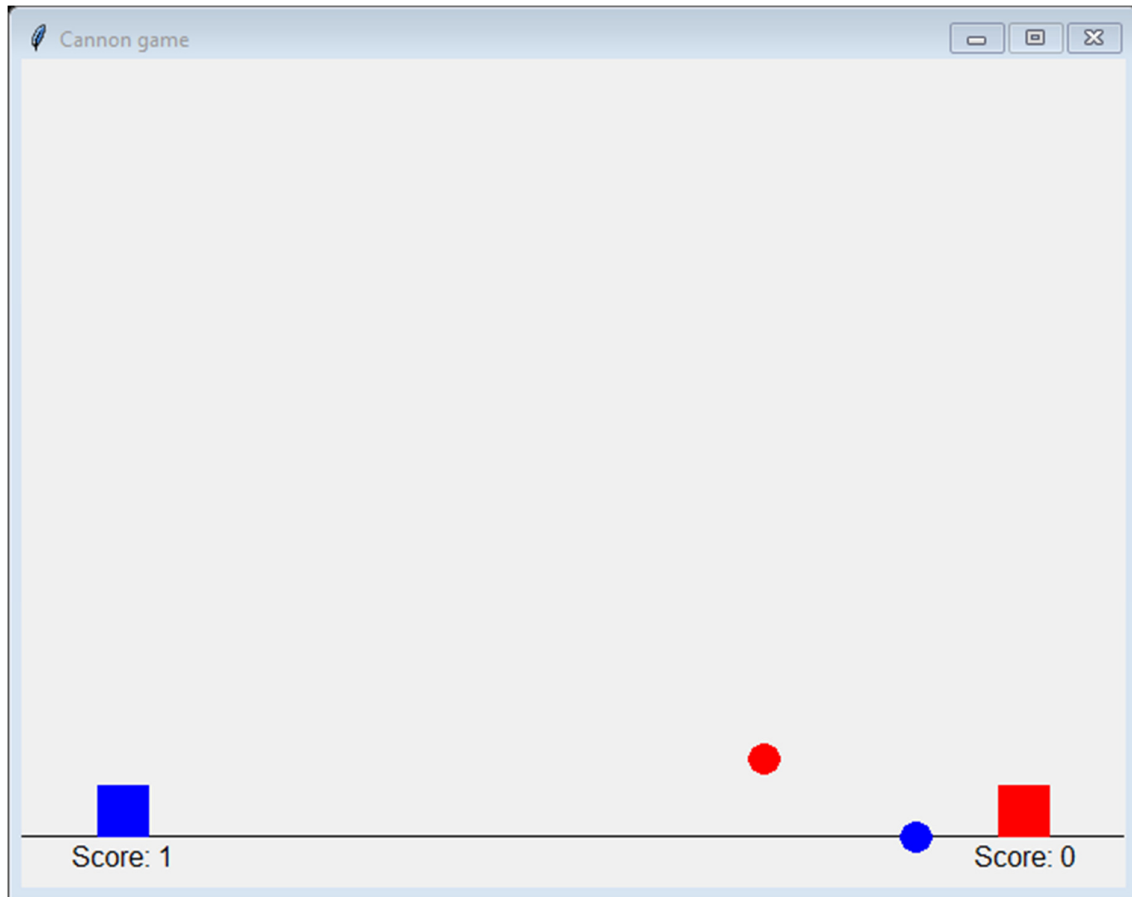
## Testing the game graphics

After testgame.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 `testgame.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 hit. 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.