Introduction to Java Programming

Mini Project: Tank Shooter

Overview

In this mini project, you and your partner will build a tank shooter game in Java. The rules of the game are simple: two players each control their own tank on a shared keyboard (the game is played on a single computer) to battle it out on a map by shooting at each other. The first player to reduce the other player's health to zero wins.

Since this is a game, we will need to work with a GUI (graphic user interface), which can include things such as buttons, labels, windows, and panels. We will also need the ability to draw in-game objects such as tanks and projectiles on the computer screen. To do this, we will use the Java Swing library. Java Swing provides programmers with a platform-independent (ie. it works on all operating systems) framework for building interactive applications without having to manually pass the objects we want to draw on the screen to the computer's graphics card—Swing automatically does that for us. With Swing, programmers can simply call a few library functions and interactive objects will then appear on the screen, almost as if by magic.

For the Tank Shooter project, you will be provided with source code that sets everything up for you, including Java Swing. Your focus is on programming the necessary game mechanics, drawing the appropriate game objects, and implementing various other features that make that gameplay more fun!

Set-up

Before starting this project, make sure you have a JDK (Java Development Kit) and an IDE (Interactive Development Environment) such as Eclipse installed. If you're having trouble installing either, follow the instructions in the link below:
http://www.ntu.edu.sg/home/ehchua/programming/howto/eclipsejava_howto.html

Once Eclipse has been installed, do the following:

1. Download the source code folder (it should be named Tank_Shooter_Base_Code)
2. Decompress the source code folder, if necessary
3. Import it into Eclipse (open up Eclipse, go to File > Import > General > Existing Projects into Workspace > Browse… > (find the source code folder) > Finish)

Once the source code folder has been imported into the Eclipse workspace, you can run it by going under the src folder inside the source code folder, right clicking on Game.java, and clicking Run As > Java Application. A full-screen window should pop up that shows two circles: a yellow circle near the top left corner of the screen, and a red circle near the bottom right corner of the screen. If you cannot see the red circle, check to see if the window is actually full-screen (particularly for Mac computers). The game can take a couple of seconds to load; this is normal.

Programming Tasks

This is a fairly complex project, so we've broken it down into a bunch of smaller, manageable features. For the final submission, you and your partner will need to implement ***all*** of the features under the **Basic** section, and ***one*** feature of your choice under the **Challenge** section. The Basic features are listed roughly in the order that you should implement them (you probably should implement the Challenge feature after you've finished with the Basic features!), but this is only a guideline, so do feel free to implement a later feature if you wish or if you're stuck on an earlier one. Each of the features below are also outlined in the source code via a `TODO`.

- **Basic Features**

1. Complete the `addKeyListener` constructor call in Arena's constructor (see Feature 2 and Feature 8 for exactly which key presses you should detect). This will tell Java Swing to listen for keyboard input, which is important since the players control their tanks using the keyboard.
    - To do this task, you need to add variable(s) to the Player class that stores key statuses (eg. a `boolean` variable called `isWPressed`, and so on), and update these variables inside the `keyPressed` and `keyReleased` functions inside the `addKeyListener` constructor.
2. Handle the following movement-related key presses in Arena's `tick()` function:
    - For the Yellow player:
        i. W: move forward
        ii. S: move backward
        iii. A: rotate counterclockwise
        iv. D: rotate clockwise
    - For the Red player:
        i. Up arrow: move forward
        ii. Down arrow: move backward
        iii. Left arrow: rotate counterclockwise
        iv. Right arrow: rotate clockwise
    - To do this task, you need to check the key statuses of the variables you created in Feature 1 above, and update the `orientation` variable for each Player if a rotation key has been pressed, or update the velocity variables (`vX`, `vY`) for each Player if a movement key has been pressed. Make sure to use the `timeDelta` variable in `tick()` to properly compute the amount to rotate/move.
3. Draw the player's barrel and health bar in Player's `draw` function. The tank's barrel should rotate around its circular body when the player presses the appropriate keys as specified in Feature 2 above. The Player's health bar should have an outline as well as a an actual filled-in part indicating the remaining health of the player.
    - To do this task, you should use the `fillPolygon` and `drawPolygon` functions to draw the barrel. You will need to pass in a set of rotated points' coordinates into the function in order to properly draw the rotated barrel. To do this, figure out the coordinates of the vertices of a non-rotated barrel (which can simply be a rectangle), and then pass these coordinates into GameObject's `rotatePoint` function, which has already been programmed for you. To draw the health bar, use the `fillRect` and `drawRect` functions, or if you want a cooler look, use the `fillRoundRect` and `drawRoundRect` functions.
4. Complete the `move` function in the Player class. The function signature passes in a `map`, a `translateX` (the amount you want to move horizontally), and a `translateY` (the amount you want to move vertically). For now, ignore the map and just focus on moving the player by the amount specified by the other two

arguments. Inside this function, you should detect whether the player has hit the edge of the arena, and if so, properly update the player's position and velocity.

- To do this task, take advantage of the functions we've already provided you in the GameObject class. If coded succinctly, your `move` function should not be more than a few lines long!

5. Move the players in Arena's `tick()` function by calling the `move` function you coded in Feature 4 above. After moving the two players, you should detect whether the players themselves have collided with each other, and if so, update the players' velocities (and positions, if you want to get a little fancy and make the result look even better). It's up to you whether you want the players to stop moving or bounce off when they've collided with each other, but the result should *not* be the two players overlapping for an extended period of time. There should also be friction, so if a player isn't moving forwards or backwards, her velocity should decrease in magnitude.
   - To do this task, take advantage of the functions we've already provided you in the GameObject class. For friction, you can simply multiply the velocities for both players by a constant between 0 and 1 (tune it so that it looks realistic), and to make the result look even more realistic, set the velocity to zero if its magnitude is less than some value, say 0.0001. This will prevent rounding issues in Java where the player's velocity never reaches 0 and thus the player continues to inch along long after you've stopped moving it.
6. Draw the bullet in PlayerProjectile's `draw` function. Inside this function, you should only draw one bullet, because we're going to call the `draw` function for each bullet separately. Bullets fired by the Red player should be red, and bullets fired by the Yellow player should be yellow. You are free to draw your bullet however you want, but we suggest that you give it a circular border for collision detection purposes later on.
   - To do this task, take advantage of the `source` variable in the PlayerProjectile class. The `source` variable tells you which Player (through the Player's `id` variable) fired that projectile. Use that information to tell you which color the bullet should be.
7. Complete the `move` function in the PlayerProjectile class. This function has a similar purpose as Feature 4 above. However, regardless of how you dealt with player-edge collisions in Feature 4, we require that you make bullets bounce off of the arena walls. Do not worry about bullet-player collisions in this function.
   - To do this task, take advantage of the functions we've already provided you in the GameObject class. If coded succinctly, your `move` function should not be more than a few lines long!
8. In Arena's `tick()` function, check for the Red player if the Space bar is pressed down, and check for the Yellow player if the Enter key is pressed down. However, there is a reload time for each tank (however fun it looks, we don't want each player to be able to shoot a continuous stream of bullets). Only when both the proper key is pressed down and the reload time has gone down to zero should you actually fire a bullet. To fire a bullet, create a new bullet using BasicWeapon's constructor (BasicWeapon is one type of weapon. PlayerProjectile is a "parent" class that is shared across all types of projectiles/bullets, which makes it much simpler to add a new weapon, if you choose to do that Challenge feature).
   - To do this task, you will need to add extra variables in the Player class to store the current and maximum reload times for the player, as well as a list of all bullets fired by that player. Make sure to initialize them in the Player constructor. We suggest using an ArrayList or a LinkedList to store the list of bullets for each player.

9. For each bullet (fired by both players), move it by calling the `move` function you coded in Feature 7 above. However, note that each bullet has a `range` variable, which indicates how far that bullet can travel. So you need to update the bullet's `distanceTravelled` variable and compare it to its `range`. If the bullet's `distanceTravelled` is less than or equal to its `range`, you should move it. Otherwise, you should delete the bullet from the list for player who owns it. After moving each bullet, you should detect if the bullet has hit the enemy tank. If so, you should update the enemy's `health` accordingly. Each bullet also has a `penetration` variable, which represents how many times it can damage the enemy. Decrement this variable by 1 every time the bullet hits the enemy. Delete the bullet once its `penetration` becomes 0.
   - To do this task, think about (mathematically) how to detect when two circles have intersected with each other (both the bullet and the player can be assumed to be circles). You do not need to use any of GameObject's functions to do this. Also, do not iterate over the list of bullets for each player and delete it at the same time: Java will give you an error! Instead, add the bullets you want to delete into a new list and use the `oldList.removeAll(newList)` function to do the removal. You do not need to worry about making the bullet bounce off players, or make the bullets subject to friction. Finally, make sure to use the `timeDelta` variable in `tick()` to properly compute the amount to move the bullet.
10. Draw all of the bullets in Arena's `paintComponent` function. You should draw the bullets before drawing the players so that they don't show up on top of the tanks.
11. At the end of Arena's `tick()` function, detect whether either player's `health` has dropped to 0. If so, the game has ended, so you should stop the timer and display a message in the middle of the screen showing who won, or if there was a tie (say, if both player's health was reduced to 0 at the same time).
   - To do this task, you should create a JLabel to display the text. To position the text on the center of the screen, do the following:
      i. Add `this.setLayout(new BorderLayout());` before creating the JLabel
      ii. Add `gameEndText.setHorizontalAlignment(JLabel.CENTER);` after creating the JLabel, where `gameEndText` is the name of your new JLabel
      iii. Add `this.add(gameEndText, BorderLayout.CENTER);` and `super.validate();` after Step ii. above
   - To get a sense of what the steps above are actually doing, you might want to Google "Java Swing BorderLayout."
12. Play around with the `INIT_` variables in Player and BasicWeapon. These variables represent the starting statistics for the players and bullets. For example, you can increase `INIT_SIZE` to make the players and/or bullets bigger, and you can increase `INIT_SPEED` to make the players and/or bullets go faster. For now, you don't have to worry about the `INIT_MASS` variables. In your final submission, these variables should be set to something reasonable.
   - These variables are a great (and fun!) way for you to test your game. For example, if increasing `INIT_SPEED` doesn't actually make your player and/or bullets go faster, then there's something wrong in your code. Similarly, setting `INIT_PENETRATION` to 2 should allow bullets to go through the first enemy tank they hit (it may hit it a second time upon bouncing back, at which point the bullet should actually disappear).

➢ Challenge Features

In addition to all of the Basic features above, you and your partner need to choose *one* of the following Challenge features to implement for your mini project. Here, *one* means one feature per partner pair.

1. **Random map generation with obstacles/terrain.**
   - Instead of having just an empty arena, the arena should be filled with obstacles. These could resemble walls or simply impenetrable land. The map should be randomly generated, meaning that every time you run the game, a different map should be generated without you having to change any code. Also, players should not be able to walk through walls/obstacles, and bullets should bounce off of walls just like how they bounce off the edges of the arena. Obviously, your map should ensure that the player does not spawn on top of an obstacle, and that there's enough empty space on the map so that the two players can actually navigate around the map and battle. For ***full*** points, your map should look nice, with a labyrinth/cave-like structure. For ***bonus*** points, you can add other terrain such as water or ice, which players and bullets can traverse but are given a speed penalty or boost when they're on it.
     - We strongly recommend you implementing this feature by using a cellular automata method via the Map class's `generateMap` function. This will be discussed in more detail during class. You will also need to update the `move` functions in the Player and PlayerProjectile classes to properly detect and handle player-obstacle and projectile-obstacle collisions. You may want to write your own collision detection function or take advantage of what's provided in the GameObject class, whichever works for you. Don't forget to create the map in the Arena class, instantiate it in Arena's constructor, and draw it in Arena's `paintComponent` function!
2. **Multiple weapons and weapon swapping.**
   - Instead of having just one weapon available to the players, add another weapon to the game (for example, a weapon that fires multiple bullets at once in an angle, a weapon that fires both forwards and backward, or anything else that your creative mind can conjure up) and allow players to be able to swap between the two weapons in-game using another key. The two weapons should have their own statistics, so you need to keep track of separate reload time variables in the Player class. For ***full*** points, your new weapon should be sufficiently different from the basic weapon. For ***bonus*** points, you can add multiple different weapons, and allow players to rotate back and forth between the different weapons, instead of just swapping between 2 weapons.
     - To create your new weapon, you will need to create a new class (File > New > Class). Follow the example in the BasicWeapon class for what to put in your new class. You will need a new variable in the Player class to keep track of the player's current weapon, and based on the value of that variable, draw the weapon appropriately in Player's `draw` function. We recommend you using an enum to refer to the different weapons (it makes your code clear and less prone to bugs). The Weapon class already creates this enum for you, so you can simply add onto this enum and use it across the rest of your code. To make weapon switching look good, you might want to implement a weapon switch delay, which is the minimum amount of time you need to wait in order to switch to the next weapon. Otherwise, pressing the weapon switching key would switch the weapon once every frame, which would be *way* too fast.
3. **Animated damage counters.**

- When a player receives damage, make a small line of text show up above the player that indicates how much damage he received. Animate this line of text, for example, so that it rises up above the player and disappears after a short time period (you can animate it however you want, but as long as damage text isn't non-moving and disappears after a certain amount of time, you'll be fine). For ***full*** points, you'll need to achieve the effect described above. There are no bonus points available for this feature.
  - ◆ We've already created the DamageCounter class for you. In it, we've provided an empty constructor and an empty `draw` function, and some more hints. Think about what information each damage counter should store so that you can draw the appropriate text at the appropriate location on the screen. Feel free to modify the arguments passed to the constructor. After completing the DamageCounter class, don't forget to create a list of damage counters in the Arena class, instantiate it in Arena's constructors, add damage counters/remove damage counters from it appropriately as the game progresses, and draw them in Arena's `paintComponent` function!

4. **Realistic collisions.**

- So far, the Basic features do not require you to simulate fully realistic collisions that follow the laws of physics (in fact, you are allowed to have the players stop moving after they collide with each other!). To make your game look even better, you should implement realistic player-player and player-bullet collisions. If we treat the players as circles, one example of what could happen in a player-player collision is as follows:

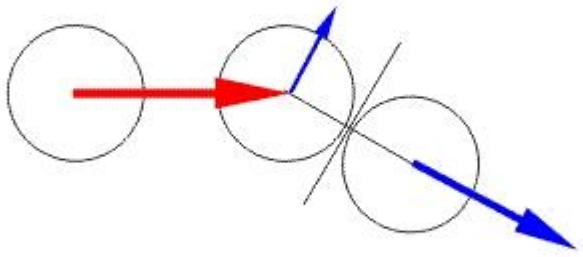

Image credit: https://courses.cs.washington.edu/courses/cse557/08wi/projects/final/artifacts/rkamin-yang/

Initially, the circle on the left starts moving in the direction of the big red arrow (think of the red arrow as its velocity). It eventually collides with the circle on the right, which is initially unmoving. At that moment, its position is at that of the circle in the center. After the collision, the two circles move apart in the directions indicated by the smaller blue arrows (the original circle on the left moves towards the top right while the original circle on the right moves towards the bottom right).

For this Challenge feature, you'll be asked to implement exactly what's show above. When two players or a player and a bullet collide, their resulting positions and velocities should be updated realistically (think of colliding billiard balls if you need another example). This means that bullets can no longer go through players: players should "absorb" some of the momentum of the bullet hitting them, and if the bullet's penetration is still positive after hitting the player, it should bounce off realistically. Also, when the player is at the edge of the arena and the player is pressing a move key, it should "roll" along the edge of the arena in any direction it's able to (for example, if the Red player at the top edge of the arena facing towards the top right and the Up key is pressed, the Red player should roll towards the top right corner of the arena and get stuck once it gets there). In other words, players should not be stuck, nor should they bounce, when they're at the edge of the arena but not at a corner. Finally, two players should *never* overlap with each other, not even for a brief moment! This should be the case even when one player is trying to squish the other player into an edge of the map. For ***full*** points, your collisions should be smooth and not involve any tremoring, shaking, or other weird position/velocity changes. For ***bonus*** points, allow the players to damage each other simply by bumping/crashing into each other, where the player being bumped into (ie. the player who gained momentum through the collision) gets dealt more damage than the player doing the bumping, with the difference scaled based on the amount of momentum gained—basically, calculate the damage realistically: a strong crash should result in more damage than a light bump.

◆ We recommend you to implement an entirely separate circle-circle collision function for this feature. We've already set up the function signature for you in the GameObject class as the `bounceWith` function. However, feel free to change the arguments for this function to suit your needs (the current function signature simply represent how we implemented it, but there are many other ways to do it as well!). After coding this function, you will likely need to update your `move` function in Player (for player-edge collisions), and the player-player and player-bullet collision handling sections of Arena's `tick()` function to either call this new function or make other appropriate edits. Since this is a fairly complex feature (at least mathematically), feel free to use the Internet to figure out what you need to do (eg. Google "circle-circle collision")!

5. **Restarting the game.**
   - Instead of just displaying a message when the game has ended, you should allow the players to restart the game as well. This means you should add a button below the game over message that when clicked, resets the arena to its initial state so that a new game can be played. To make this message-button combination look nice, you should explore the Java Swing library to see what features there are. For example, you could put both the message and the button inside a bordered panel, with eye-catching colors and good layout/formatting. For ***full*** points, your restart panel should not only work, but also look good. There are no bonus points available for this feature.

     ◆ We recommend you looking into Java Swing's GridBagLayout to create your panel and layout the message and button. If you choose to do this, Step 11i. of the Basic features above will need to be replaced with `this.setLayout(new GridBagLayout());`. You should also Google the Java Swing documentation on GridBagLayout—there are many helpful tutorials out there! Also, don't forget to restart the timer and call the `reset()` function in Arena when you restart the game.

Advice

As you can probably tell from the programming tasks above, this project is *not* trivial! You will likely end up writing hundreds of lines of code, if not over a thousand. We give you this document so that you know exactly what you need to implement, and guidelines on how to do so. Make sure to start the day this project is assigned, and pace yourself. Implement a Basic feature or two a day (don't forget to debug!), and make sure to spend ample time on the Challenge feature that you and your partner decide to do. This is *not* the type of project that you can procrastinate until the weekend before it's due and stay up late to finish it. A good programmer spreads out his/her work over time!

As you code, you will inevitably find bugs in your code. When you debug, always keep in mind what you're trying to achieve (remember back to the in-class demo when the project was first assigned). If you don't know what's causing your bug, print out relevant variables to the Console (remember our friend `System.out.println(whatYouWantToPrint)` ?) at critical places in your code. Analyze the values of those variables. Do they make sense given their values at that particular place in your code? If not, something earlier caused this faulty value to appear. Otherwise, the bug is further down in execution. Once you've tracked down exactly where in your code the bug is located, try to analyze the actual code. Does your code make logical sense? If you can explain your code back to yourself (or to your partner), in complete sentences, with no logical flaws, that should make you more confident about the correctness of your code. If any of this seem vague to you, we will also spend some time in class specifically for the purpose of teaching you how to debug your code.

If you're hitting a road block, say a feature you don't know how to implement, or a bug you don't know how to fix, make sure to reach out for help! Ask your classmates, your parents, and your instructor (via email or in person). Your instructor should be able to assist you outside class (say, during lunch/break times) provided you come prepared with question(s) to ask and/or bug(s) to fix.

Since this project is done in pairs, you will need to either code alongside your partner, or code separately and integrate together later on. Either approach is fine, although we strongly suggest you agree on a division of labor early on, and in such a way that you don't end up overwriting each other's implemented functionality (ie. working on the same functions in the same files). If you partition the tasks well, you should not need to spend much time figuring out how to integrate your code together. Also, make sure to work together, a lot! A lot of the Basic features build on top of each other, so you cannot really test a later feature if an earlier feature is broken. Work together with your partner frequently so that you can integrate your code step by step (after each of you have implemented a feature each, for example), then test, debug and make sure everything's working, and *only then* move onto the next couple of features. You will also likely need to work closely together to implement the Challenge feature, since it's fairly involved. Before the final demo, make sure to get together with your partner, integrate everything, debug everything, and have a single, final working version ready for submission.

Proposed Schedule & Suggested Implementation Timeline

- Wednesday, January 9th, 2019
  - In-class: Overview of source code; introduction to Java object-oriented design; discussion of game-programming paradigms/Java Swing; getting started on Basic features 1 and 3
  - Homework: Fully understand the code base and complete Basic features 1, 2, and 3
- Thursday, January 10th, 2019

    - In-class: Debugging workshop; discussion of player physics; introduction to Java inheritance/overriding; getting started on Basic features 4 and 5
    - Homework: Complete Basic features 4, 5, 6, and 7
- Friday, January 11th, 2019
    - In-class: Getting started on the Challenge features with special focus on random map generation; discussion of Java data structures (ArrayList, LinkedList, HashMap), static/final, and encapsulation
    - Homework: Complete Basic features 8, 9, 10, 11, and 12, along with the Challenge feature
- Monday, January 14th, 2019
    - In-class: Project demos

Submission

In addition to the in-class demo, you will also need to submit your final code to your instructor, who will evaluate it for completeness and correctness. Make sure to submit the code via email, along with any relevant other documents (eg. attribution…etc.). Furthermore, you will need to submit a short team evaluation at the end of class on Monday, January 14th, 2019. More details will be given in class.

Evaluation

Your final mark will be based on a combination of individual in-class participation, team evaluation, as well as your final demo and code submission. Following is the rubric on which you will be scored:

Basic features	10 marks each × 12 = 120 marks.
Challenge feature	50 marks
Team evaluation	10 marks
In-class participation	20 marks
Total	200 marks

Bonus marks (up to 30 marks) can also be earned: 3 of the 5 Challenge features offer them (see above).

Attribution

While we encourage you to seek out help as you work on this project, we realize that it's often very tempting to just copy some lines of code here and there from the Internet or from another outside source and claim it as your own. We *strongly discourage* this. However, if you do find the necessity to look up how to implement certain things from an outside source, *please* make sure to cite it. So, if you used code from a link online or from a book, copy the link's URL or the book's title & relevant page numbers into a text file (.txt). Your final code submission should contain a .txt document named attribution.txt that either contains a list of citations of the outside resources you used or the sentence "My partner and I did not use any outside sources for this project" along with the names of your partner and you. Do not forget to submit this document!

That being said, there are many helpful websites out there: https://stackoverflow.com/ is a great example. Also, feel free to look up Java documentation at any time—they're all available at https://docs.oracle.com/javase/8/docs/api/index.html