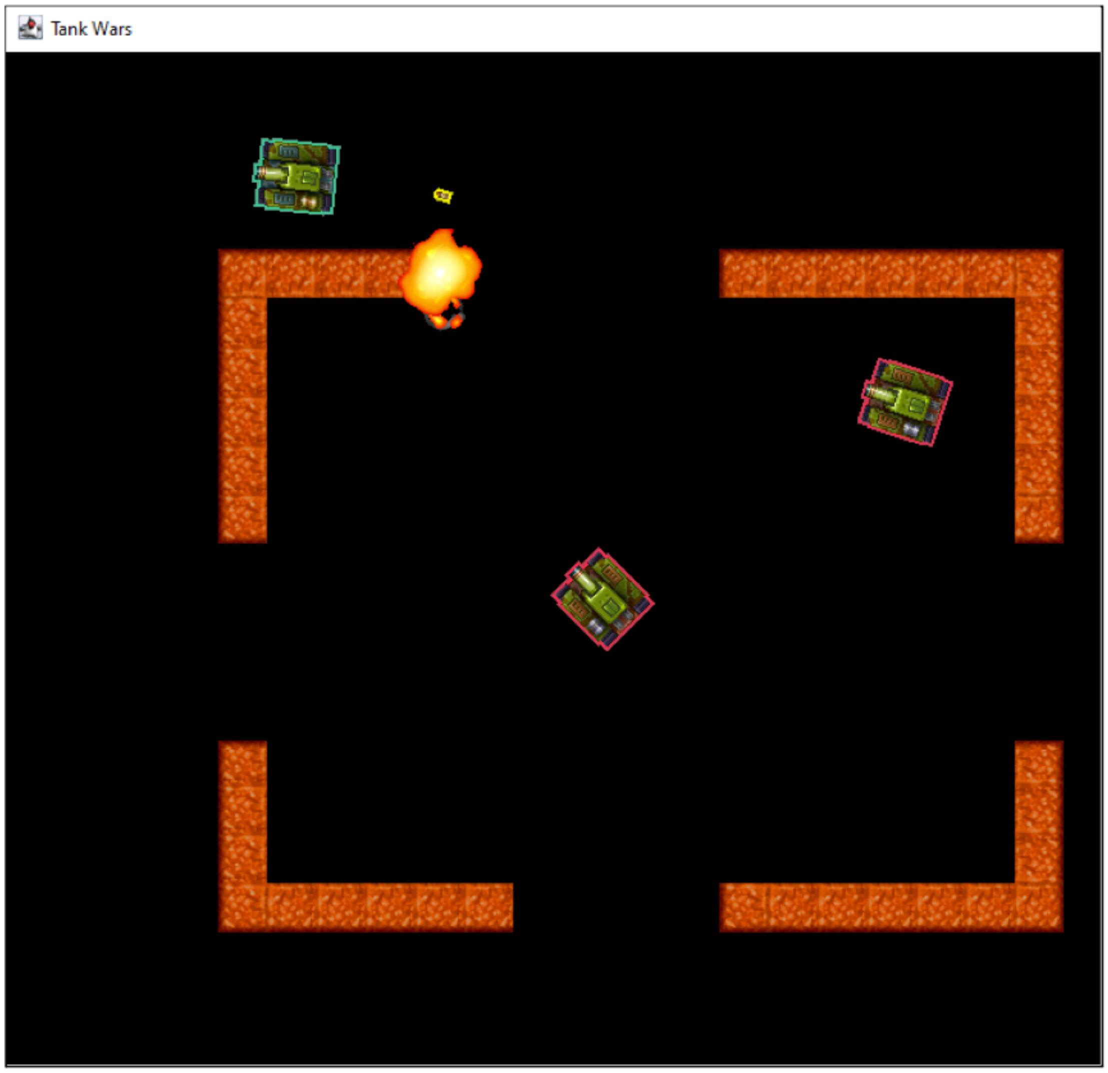
San Francisco State University
CSC 413: Software Development
Dawson Zhou (zhoud@sfsu.edu)

Term Project

Overview

The goal of the term project is to apply design principles we've learned throughout the semester to develop a simple video game with a focus on extensibility.



Detailed Requirements

Part 1 Requirements

Part 1 is due by **Tuesday, May 4, 2021** at **11:59 pm PT**. In order to receive full credit for Part 1, you must submit the following features to your GitHub repository:

- A player tank and an AI tank show on the screen at the same time.
- The player tank moves based on keyboard input.
- The player tank shoots shells in response to keyboard input.
- The AI tank moves according to some logic, independent of the player tank.

See the next few sections for details on how to approach implementing these features.

You don't need for these features to be working perfectly by the Part 1 deadline to receive full credit, but you must show a good faith effort in your attempt. Please note that you will only receive credit for Part 1 if you commit your change(s) and push them to GitHub.

Understanding the Starter Code

The first major step to tackling the project is to spend some time reading over the provided starter code to understand what is already implemented, and what classes you'll need to interact with.

The classes provided include the following:

- **GameDriver.java**: The controller of our game. This is where the main method is defined as the entry point of the program. `GameDriver` defines the gameplay loop which drives the game. You are responsible for implementing the `setUpGame`, `updateGame`, and `resetGame` methods.

- **KeyboardReader.java**: The class responsible for understanding keyboard input. The `KeyboardReader` is provided to you as a singleton class. You can ask its instance if a key is currently being pressed. You won't need to modify this class, unless you decide to add an extra feature that requires additional key input.

- **WallInformation.java**: A convenient way to package information you need about walls. See Walls below for more details on how to use the class.

- **Constants.java**: Various useful constant values.

- **model/Tank.java**: A basic version of a model class that can be used to represent tanks. You will need to add classes for other entity types, and find ways to reduce code duplication between them.

- **model/GameWorld.java**: The class tracking all entities present in the game world. You will be defining and implementing methods so that any other code in the project which needs to access the various entities in the game world can do so.

- **view/RunGameView.java**: The view class that you'll be interacting with to draw images on screen. You won't need to modify this class, but you will be calling its various public methods.

- **view/**<other files>: Other classes controlling the UI aspects of the game. This code is provided to you, and won't need to be modified.

Tanks

Every entity in the game will have two aspects to its representation: the model object, and the view sprite. The model object will be defined by a class in the **edu.csc413.tankgame.model** package, and the corresponding sprite in the view will share its String id.

To start with, focus on adding two tanks to the game. The `GameWorld` will track the tank model objects, while the `RunGameView` will track their corresponding sprites. Once you're able to display the two tanks on screen, work on implementing movement logic: one as a player tank responding to keyboard input, and one as an AI tank determining its own movement logic.

See Lecture 24 for more details.

Shells

Once you have tanks moving around on screen, you can add the ability for tanks to fire shells. There are a few key observations that will help us determine how to add shells:

- Player tanks fire shells based on keyboard input (when the spacebar is pressed), while AI tanks fire shells based on their movement logic.
- When a shell is created, its starting position and angle are based on the tank that fired it.
- A shell moves in a straight line, maintaining a constant angle, until it goes off screen.

A design that suits all of the requirements is to define a **`Shell`** class for shell entities which tracks their position and angle. A shell can move each iteration of the gameplay loop, just like a tank, so you will want to find a way to share code between the two -- I strongly recommend defining an abstract **`Entity`** class which contains code common to both.

When a tank shoots a shell, a **`Shell`** object needs to be created and added to the **`GameWorld`**. Since tanks determine when to shoot shells in their **`.move(...)`** methods, they will need to have access to the **`GameWorld`** so they can create and add the **`Shell`**. Keep this in mind when implementing the **`.move(...)`** method for entities: you will need to pass the **`GameWorld`** in as a parameter.

There's an additional problem when a shell is fired -- even once we've found a way to add the new **`Shell`** entity to the **`GameWorld`**, it won't have a corresponding sprite added to the **`RunGameView`**, so we won't see its image on screen. We don't want the tank **`.move(...)`** logic to be responsible for adding the sprite as well, because it violates the Separation of Concerns principle: model logic shouldn't be directly dealing with view logic. What we can instead do is record in the **`GameWorld`** that there is a brand new **`Shell`** entity, and a corresponding sprite needs to be added as well. After all existing entities are done moving, the main gameplay loop in **`GameDriver`** can then add the corresponding sprites for all of those new shells.

See Lecture 25 for more details.

GameWorld Aware AI Tank Logic

You are required to have at least two different types of AI tanks, with both being on screen at the same time. At least one of those AI types must base its **`.move(...)`** logic on information about the **`GameWorld`**, such as the player's current location.

The following is some logic I used during in-class demos that will cause an AI tank to always be turning so that they face the player tank. You may feel free to use this code within your **`.move(...)`** logic for a GameWorld aware AI tank.

```java
Entity playerTank = gameWorld.getEntity(Constants.PLAYER_TANK_ID);

// To figure out what angle the AI tank needs to face, we'll use the
// change in the x and y axes between the AI and player tanks.
double dx = playerTank.getX() - getX();
double dy = playerTank.getY() - getY();
```

```
// atan2 applies arctangent to the ratio of the two provided values.
double angleToPlayer = Math.atan2(dy, dx);
double angleDifference = getAngle() - angleToPlayer;

// We want to keep the angle difference between -180 degrees and 180
// degrees for the next step. This ensures that anything outside of
that
// range is adjusted by 360 degrees at a time until it is, so that the
// angle is still equivalent.
angleDifference -=
        Math.floor(angleDifference / Math.toRadians(360.0) + 0.5)
                * Math.toRadians(360.0);

// The angle difference being positive or negative determines if we
turn
// left or right. However, we don't want the Tank to be constantly
// bouncing back and forth around 0 degrees, alternating between left
// and right turns, so we build in a small margin of error.
if (angleDifference < -Math.toRadians(3.0)) {
    turnRight(Constants.TANK_TURN_SPEED);
} else if (angleDifference > Math.toRadians(3.0)) {
    turnLeft(Constants.TANK_TURN_SPEED);
}
```

If you do use the above code in your implementation of an AI tank, you must add some additional movement logic of your own to receive full credit.

Bounds Checking

We need bounds checking to deal with tanks and shells going off screen. It's fairly straightforward to determine if an entity is off-screen -- we just check if its x coordinate is less than the minimum x allowed or greater than the maximum x allowed for that entity, and likewise for its y coordinate.

The minimum and maximum x and y values allowed are provided for tanks and for shells in the **`Constants`** class. The type of entity determines what should happen next if the entity is found to be off-screen.

- A tank that is off-screen should have its location updated so that it is back on-screen.
    - If the x coordinate is less than `Constants.TANK_X_LOWER_BOUND`, it should be set to `Constants.TANK_X_LOWER_BOUND`.
    - If the x coordinate is greater than `Constants.TANK_X_UPPER_BOUND`, it should be set to `Constants.TANK_X_UPPER_BOUND`.
    - If the y coordinate is less than `Constants.TANK_Y_LOWER_BOUND`, it should be set to `Constants.TANK_Y_LOWER_BOUND`.
    - If the y coordinate is greater than `Constants.TANK_Y_UPPER_BOUND`, it should be set to `Constants.TANK_Y_UPPER_BOUND`.
- A shell that is off-screen should be removed from the `GameWorld`.
    - To receive full credit, the shell's corresponding sprite must also be removed from the `RunGameView`. To do this, when you remove the shell from the `GameWorld`, you'll need to "remember" it later on so that you can also remove its sprite from the `RunGameView`.
    - Tracking which shells need to be removed can also make it simpler to add animations such as a small shell explosion to the `RunGameView` when the shell is removed.

Limiting Shells

If you've implemented support for tanks shooting shells, but without any limits, you may have noticed that tanks will fire a huge number of shells in a stream -- a feature that looks fun, but is not particularly good for a playable game. You'll need to add a limit so that tanks are not able to fire a shell on every single iteration of the gameplay loop.

There are a number of potential approaches here. Here are two simple suggestions:

- A tank must wait 200 (as an example) iterations since the last time it fired a shell before it can shoot again. You can track this by storing a "cooldown" integer which starts at 200 and is decremented for each call to `.move(...)` until it hits 0, at which point it can shoot again. When a shell is fired, the cooldown is reset to 200.
- Each tank can only have one shell on screen at a time. You'll need to implement a way to track existing shells for a tank to determine if the tank is allowed to shoot.

Walls

If you have designed your `Tank` and `Shell` classes to share code, adding walls should be fairly straightforward -- it's just another game entity with a location and an angle (always zero degrees), and it doesn't do anything when asked to move, turn, or check bounds. Walls will also be involved in collision detection like other entity types.

I've provided a **WallInformation.java** file in the starter project, which reads from a walls.txt file in the "resources" folder. The walls.txt file treats the game world as a grid of integers, with each integer indicating if there should be a wall at that location, and if there is, what that wall should look like. `WallInformation` has a static method, `.readWalls()`, which will read the text file in and convert it to a `List<WallInformation>` representing all of the walls that need to be added to the game. Each individual `WallInformation` in the list will have an associated image file name, accessible via `.getImageFile()`, and x and y coordinates for the wall.

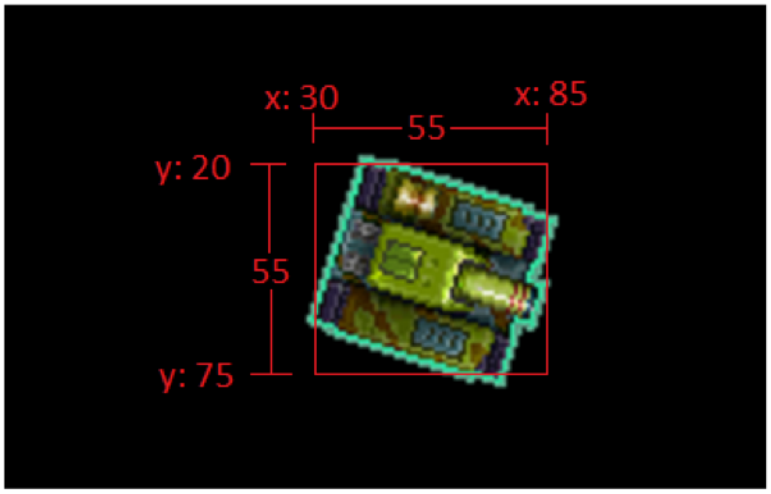
You will be responsible for creating a new class to represent wall entities in the game model, similar to tanks and shells. Walls should also have a unique ID, a location, and an angle of 0.

Collision Detection

A major step in adding depth to your game is supporting collision detection between game entities. Collision detection will prevent tanks from driving through each other and through walls, and will allow shells to interact with the environment rather than passing through everything.

The first step for supporting collisions is determining which pairs of entities are "colliding" at every step of the game. This can be made simpler by treating every entity as a rectangle oriented along the x and y axes, rather than taking the exact entity shape and rotation into account. It isn't as accurate as using the entity's exact geometry, but it is much simpler to implement and very efficient to run.

The entity's rectangle stretches from the entity's (x, y) location in its top left corner to its bottom right corner. We'll call this the **bounding box**. For example, consider the following tank:



In this example, the tank is located at x: 30, y: 20 with a width of 55 and a height of 55. That means that its **x bound** (i.e. the right side of the bounding box) is at x: 85, and its **y bound** (i.e. the bottom side of the bounding box) is at y: 75.

If we have the bounding boxes for two entities, we can determine if they're colliding by asking the inverse: how would we check that they're ***not*** colliding? To answer that, we look at the x axis and y axis separately.

- If the first box's left side is to the right of the second box's right side OR
- If the first box's right side is to the left of the second box's left side

...then the two boxes are not overlapping at any point along the x axis.

- If the first box's top side is below the second box's bottom side OR
- If the first box's bottom side is above the second box's top side

...then the two boxes are not overlapping at any point along the y axis.

If any of the conditions above is true, then the two bounding boxes do not overlap. Otherwise, they do. Translated to x coordinates, x bounds, y coordinates, and y bounds, we check if:

- First box's x coordinate > second box's x bound OR
- First box's x bound < second box's x coordinate OR
- First box's y coordinate > second box's y bound OR
- First box's y bound < second box's y coordinate

Any of these being true implies that the boxes do not overlap.

I would suggest adding support for your Entity class to return the x bound and y bound with methods such as `.getXBound()` and `.getYBound()`. These are dependent on the width and height of each entity, which are represented in **Constants.java**. For example, a tank's `.getXBound()` should then return `getX() + Constants.TANK_WIDTH`. With each entity type defining `.getXBound()` and `.getYBound()`, we can define a method `.entitiesOverlap()` which takes in two `Entity` objects as parameters and returns the result of the following:

- First entity's x coordinate < second entity's x bound AND
- First entity's x bound > second entity's x coordinate AND
- First entity's y coordinate < second entity's y bound AND
- First entity's y bound > second entity's y coordinate.

Note that this is inverted from the check above, since we want the method to return true if the two entities ***do*** overlap.

Once you have implemented the logic for determining if a pair of entities is colliding, you can apply that method to every single pair of entities in the `GameWorld`. Be careful on how you approach this -- a sensible idea is to iterate through each pair is to use a for loop within a for loop, but doing so will also lead self collisions (each entity "colliding" with itself) and duplicate collisions (we handle entity A colliding with entity B, and then we handle entity B colliding with entity A again). You'll need to skip those cases.

Collision Handling

Once we've detected that a pair of entities has collided, we need to determine how to handle the collision based on the entities' types. The different potential scenarios include:

- A tank colliding with a tank
- A shell colliding with a shell
- A shell colliding with a tank
- A tank colliding with a wall
- A shell colliding with a wall

You can check which scenario occurred using `instanceof`:

```java
private void handleCollision(Entity entity1, Entity entity2) {
    if (entity1 instanceof Tank && entity2 instanceof Tank) {
        // ...
    } else if (entity1 instanceof Tank && entity2 instanceof Shell) {
        // ...
    } else if (entity1 instanceof Shell && entity2 instanceof Tank) {
        // ...
    }
}
```

A Tank Colliding with a Tank

When two tanks collide, we need to update both their locations so that they don't overlap, as we don't want to allow the tanks to pass through one another. Let's label the tanks as **Tank A** and **Tank B**.

The approach we'll take is to determine which axis (x or y) and direction the tanks should move that minimizes their move distance. For the smoothest and most predictable result, we'll have both tanks move an equal distance in opposite directions.

To determine the axis of movement, let's pretend for a moment that Tank B will be anchored in place while Tank A does all the moving. In that case, the four possible moves are:

1. Tank A moves to the left until the tanks no longer overlap along the x axis. Tank A's right side (x bound) must be less than Tank B's left side (x coordinate). The distance of movement is **`tankA.getXBound() - tankB.getX()`**.
2. Tank A moves to the right until the tanks no longer overlap along the x axis. Tank A's left side (x coordinate) must be greater than Tank B's right side (x bound). The distance of movement is **`tankB.getXBound() - tankA.getX()`**.
3. Tank A moves upward until the tanks no longer overlap along the y axis. Tank A's bottom side (y bound) must be less than Tank B's top side (y coordinate). The distance of movement is **`tankA.getYBound() - tankB.getY()`**.
4. Tank A moves downward until the tanks no longer overlap along the y axis. Tank A's top side (y coordinate) must be greater than Tank B's bottom side (y bound). The distance of movement is **`tankB.getYBound() - tankA.getY()`**.

Calculate each of these four distances. The shortest of these four distances determines how we then adjust the two tanks:

1. If **`tankA.getXBound() - tankB.getX()`** is the smallest distance, then we move Tank A to the left by half that distance and Tank B to the right by half that distance.
2. If **`tankB.getXBound() - tankA.getX()`** is the smallest distance, then we move Tank A to the right by half that distance and Tank B to the left by half that distance.
3. If **`tankA.getYBound() - tankB.getY()`** is the smallest distance, then we move Tank A upward by half that distance and Tank B downward by half that distance.
4. If **`tankB.getYBound() - tankA.getY()`** is the smallest distance, then we move Tank A downward by half that distance and Tank B upward by half that distance.

Remember that to move a tank to the left, we subtract from its x coordinate; to move a tank to the right, we add to its x coordinate; to move a tank upward, we subtract from its y coordinate; and to move a tank downward, we add to its y coordinate.

A Shell Colliding with a Shell

Handling collision between shells is much simpler. They both get removed from the game. Remember to apply the same approach you did for removing shells when bounds checking so that their corresponding images are also removed from the **`RunGameView`**.

A Shell Colliding with a Tank

When a shell collides with a tank, the shell should be removed and the tank should lose a health point. To properly implement this, you'll need to add the ability for tanks (as well as walls; see below) to track how many health points they have remaining. If that value reaches zero, then the tank should also be removed from the game. If that tank happens to be the player tank, or the last AI tank, then the game should be considered over, and should transition to the end menu screen.

A Tank Colliding with a Wall

The logic for handling a tank colliding with a wall is very similar to the logic described above for a tank colliding with another tank. The difference is that only the tank will be moved the full distance needed so that they are no longer overlapping; the wall should never move.

You can calculate the same four distances (tank moving to the left, tank moving to the right, tank moving upward, and tank moving downward), pick the smallest one, and move the tank in that direction by the full distance (instead of moving both entities half of the distance).

A Shell Colliding with a Wall

Destructible environments are interesting! You can implement a basic version of this with your collision handling logic when a shell hits a wall. The shell should be removed, and the wall should lose a health point. Walls will need to track their total health points just like tanks, and when a wall runs out of health points, it should be removed from the game.

Extra Features

To get full credit for the assignment, you'll need to add some extra features to the game. I've provided a number of ideas below; if you have your own ideas, feel free to incorporate them into the game!

Extra features will be categorized as:

- **small**: worth 3 points
- **medium**: worth 6 points
- **large**: worth 10 points

If you add your own feature not listed below, feel free to reach out to me, and I can let you know if it would be counted as a small, medium, or large feature.

To receive full credit, you will need to pick and choose features totalling 15 points. If you implement more than that, you can get up to 10 additional points of extra credit.

Complex AI Logic (**medium**)

Add a complex AI tank that involves awareness of the game world in determining its move behavior. You'll need to add something beyond what I provided in this handout, but feel free to build upon that idea. Some ideas:

- AI tank that "leads" the player tank by pointing and shooting shells not directly at but some distance in front of the player tank. It should take into account how far away the player tank is, and what direction it's moving.
- AI tank that avoids shells by finding the nearest shell heading towards it in the game world, turning perpendicular to its current direction, and moving forward/backward to get out of its path.

Power Ups (**small** or **medium**)

Add power up items to the game that the player tank can pick up. I've provided an image asset for a power up icon you can display in game. Here are some ideas:

- If the player tank collides with the power up, you can treat that "collision" as the player picking it up. The result of that collision can modify the player tank's state to change some aspect of its behavior.
- If the modification is something simple, like increasing tank speed, increasing rate of fire, etc., this will be counted as a **small** extra feature.
- If the modification is more complex, like changing the behavior of shells fired (like implementing a `HomingShell` as a subclass of `Shell` which turns and tracks a target tank), this will be counted as a **medium** extra feature.

Better Collision Detection (**medium**)

The collision detection algorithm described above is a brute-force approach. We look at every single pair of entities and check if they overlap, even if two entities have no possible way of overlapping. This can potentially be very inefficient as the number of entities in the game increases (e.g. as we add walls, more tanks, more shells, etc.).

There are ways to implement a more efficient collision detection algorithm. One simple example would be to split the entire game world into a grid of boxes, each being e.g. 100 pixels by 100 pixels. Every entity is located in anywhere from 1 to 4 of these boxes. In order for two entities to possibly be colliding, they must be in the same grid box.

The collision detection algorithm would be as follows:

1. For each entity, determine which of the 100 by 100 grid boxes it belongs to. Assign the entity's ID to all of those boxes in some data structure representing the grid.
2. For each of the 100 by 100 grid boxes, perform the brute-force collision detection algorithm between all pairs of entities, but only for entities in that specific grid box.
3. Be sure to avoid detecting the same collision between the same pair of entities twice, as both entities might appear in multiple grid boxes.

The idea here would be to reduce the pairwise combinations you'd need to check, thereby preventing the time complexity of collision detection from increasing at a quadratic rate. Feel free to research and implement any other collision detection algorithms as well!

Extensible Design for Adding Collision Handlers (**large**)

Something you may experience when adding logic for handling collisions is that there's a lot of messiness to adding branches of logic for determining collision behavior based on all of the different entity types in the game. For example:

```java
private void handleCollision(Entity entity1, Entity entity2) {
    if (entity1 instanceof Tank && entity2 instanceof Tank) {
        // ...
    } else if (entity1 instanceof Tank && entity2 instanceof Shell) {
        // ...
    } else if (entity1 instanceof Shell && entity2 instanceof Tank) {
        // ...
    }
}
```

The number of cases here can quickly get out of hand, and they are hard to verify for correctness at a glance. This is a great opportunity to apply the Strategy Pattern to clean things up. The logic that handles the collision between two entities can be treated as a strategy, and we would effectively want multiple strategies to be maintained and applied when appropriate.

Consider creating a class hierarchy of `CollisionHandler` strategies, where specific subclasses (e.g. `TankTankCollisionHandler`, `TankShellCollisionHandler`, etc.) implement a common interface but with logic tailored to each specific type of collision.

Be warned: this is listed as a large feature specifically because it's challenging to design correctly in a way that actually improves the original code. To get full credit, you'll need to be thorough.