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

Term Project: Part 2

Overview

The goal of the term project is to apply design principles we've learned in class to develop a simple tank video game with a focus on extensibility for additional features. This document is a continuation of Part 1, and details expectations for your final submission for the term project.

Submission

Part 2 of the term project is due on **Wednesday, December 16, 2020** at **11:59 pm PT**. There will be no late submissions accepted for the term project.

Supplemental Resources

I've provided some additional assets (images, text files) that can be included in your resources folder, updated versions of **RunGameView.java** and **DrawableEntity.java**, and brand new **Animation.java**, **AnimationResource.java**, and **WallImageInfo.java** files. You can download them on iLearn and update your project to have access to them for Part 2.

Grading

100 pts total:

- 15 pts: Part 1 submission
- 85 pts: Part 2 submission
    - 10 pts: View elements
        - 2 pts: Displays a start menu, game screen, and end menu
        - 2 pts: Buttons are properly hooked up to listeners to transition between screens
        - 3 pts: View is kept separate from the details of the game model
        - 3 pts: Misc. e.g. DrawableEntity objects added and removed appropriately, not creating multiple windows when running the game, etc.
    - 2 pts: KeyListener handles keyboard input with appropriate scoping (not relying on public global variables, etc.)
    - 3 pts: Must transition the game to show the end menu screen if:
        - Escape key is pressed while on the run game screen
        - Player tank is destroyed
        - All AI tanks are destroyed
    - 40 pts: Model elements for game entities
        - 5 pts: Supports a player tank that moves based on keyboard input; movement speed must be limited by gameplay loop (i.e. a standard move every 8 milliseconds)

        - 7 pts: Implementation of **at least 2** different types of AI tanks with support for multiple tanks in the game at once with independent tank movement logic
            - At least one of the AI tank types must determine its behavior on the GameState, e.g. where the player is, where other tanks are, where shells on the screen are, etc. You are welcome to use my **CushionAiTank.java** logic from lecture.
        - 3 pts: Performs bounds checking on tanks to prevent them from moving offscreen
        - 10 pts: Allows the player tank and the AI tanks to shoot shells, which are displayed on screen and tracked while they move forward
            - Shells must be removed once they collide with an object or go off screen
            - Shells must be limited in how often they can be fired (as an example, it must be 200 frames since the last time you shoot a shell before you can shoot another one)
        - 5 pts: Supports adding walls in the game which block tanks and shells from passing through
        - 5 pts: Tank classes set up to share code between player tank, AI tanks, and shells
        - 5 pts: Tanks and walls track their "health", and are destroyed after taking too many hits from shells
    - 15 pts: Collision detection and handling
        - 5 pts: Supports collision detection (determining all pairs of entities that have collided with one another)
        - 10 pts: Supports collision handling (follow-up action once collision is detected, based on the types of entities colliding)
            - **Two tanks**: tanks should not pass through each other; they should each move the same distance away from each other (x or y, depending on which is closer) so they don't overlap
            - **Two shells**: both shells should be removed from the model and the view
            - **A tank and a shell**: the shell should be removed from the game, the tank should have one of its health points removed, and if the tank runs out of health points, it should also be removed
            - **A tank and a wall**: the wall should stay in place and the tank should have its x or y coordinate adjusted to move it the minimum distance so it is no longer overlapping the wall

        - **A shell and a wall**: the shell should be removed from the game, the wall should have one of its health points removed, and if the wall runs out of health points, it should also be removed
    - 15 pts: Additional features (choose **two** options, see Extra Features below)
        - Extra credit available for implementing more than two options, up to 10 pts extra.

Detailed Requirements

Details related to Part 1 of the Term Project were covered in Lectures 25 and 26. For the rest of the Tank Game features, I recommend approaching them in the order detailed below.

Game State Aware AI Tank Logic

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

I briefly showed the code for "Cushion AI Tank" in lecture 25. I'll provide the "always face the player" logic here so that you can incorporate that into your own AI Tank implementation. This can go in the AI tank's **move()** method.

```
Entity playerTank = gameState.getEntity(GameState.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();
} else if (angleDifference > Math.toRadians(3.0)) {
    turnLeft();
}
```

Bounds Checking

The Tank Game needs bounds checking to handle what happens when a tank (player or AI) or a shell goes 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. If any of these are the case, then the entity is off-screen.

The minimum and maximum x and y values allowed are provided for tanks and for shells as public constants in the **GameState** 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 **GameState.TANK_X_LOWER_BOUND**, it should be set to **GameState.TANK_X_LOWER_BOUND**.
    - If the x coordinate is greater than **GameState.TANK_X_UPPER_BOUND**, it should be set to **GameState.TANK_X_UPPER_BOUND**.
    - If the y coordinate is less than **GameState.TANK_Y_LOWER_BOUND**, it should be set to **GameState.TANK_Y_LOWER_BOUND**.
    - If the y coordinate is greater than **GameState.TANK_Y_UPPER_BOUND**, it should be set to **GameState.TANK_Y_UPPER_BOUND**.
- A shell that is off-screen should be removed from the GameState.
    - To receive full credit, the shell's corresponding DrawableEntity must also be removed from the RunGameView. To do this, when you remove the Shell from the GameState while bounds checking, you'll need to "remember" it later on in **GameDriver.update()** so that you can similarly remove it 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 -- which can be fun, but not particularly good for an actually playable game. You'll want to add a limit so that tanks are not able to shoot a shell every single time their **move()** method is called (approximately once every 8 milliseconds).

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

- A tank must wait 200 (as an example) frames since the last time it shot 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 shot, 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 common 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. Once collision detection is added, having walls will add another level of dynamic gameplay to the Tank Game.

I've provided a **WallImageInfo.java** file as a new class on iLearn. It should go in the "src/java/edu/csc413/tankgame" directory (alongside GameDriver.java). I've also provided a **walls.txt** file and several **wall-x.png** images on iLearn, which should be placed in your "resources" directory.

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. **WallImageInfo** has a static method, **readWalls()**, which will read the text file in and convert it to a List<WallImageInfo> representing all of the walls that need to be added to the game. Each individual WallImageInfo in the list will have an associated image file name, accessible via **getImageFile()**, and x and y coordinates where the image should be drawn.

You will be responsible for creating a new class to represent Walls in the game model, similar to Tanks and Shells. Just like with other entities, walls should have a unique ID, as well as a location and angle (though that angle should always be 0).

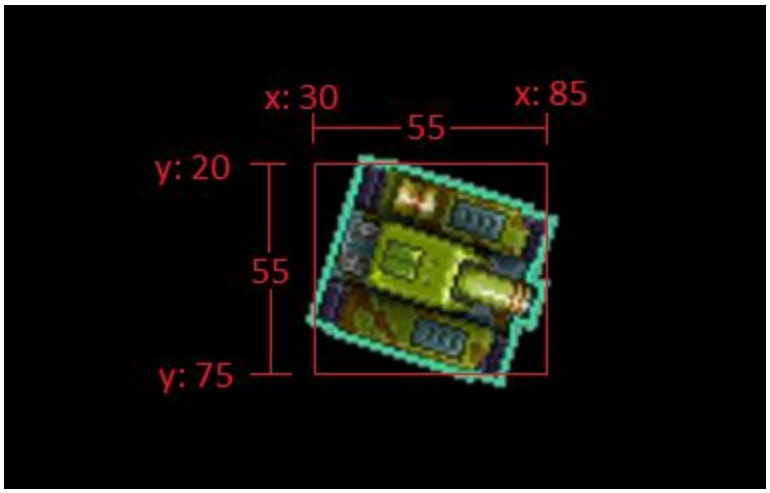
Once all of the walls have unique IDs generated for them and are added to the GameState, you should also add their corresponding images to the RunGameView via **addDrawableEntity(...)** to display them on the screen.

Collision Detection

A major step in adding interactivity to your game is detecting collisions between different 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. It will also eventually allow for actual gameplay, where tanks can take a number of hits from shells before they are destroyed.

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:



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:

- Tank: 55.0 by 55.0
- Shell: 24.0 by 24.0
- Wall: 32.0 by 32.0

A tank's **getXBound()** should then return getX() + 55.0, while a shell's **getYBound()** should return getY() + 24.0. Once each entity has **getX()**, **getXBound()**, **getY()**, and **getYBound()**, we can write the following logic which evaluates to true if two entities **do not** overlap:

```
return entity1.getX() > entity2.getXBound()
```

```
    || entity1.getXBound() < entity2.getX()
    || entity1.getY() > entity2.getYBound()
    || entity1.getYBound() < entity2.getY();
```

A more readable method for collision detection would be to write one that returns true if the entities **do** collide. We can find that by taking the logical equivalent of inverting the entirety of the previous boolean expression, which involves changing the "OR"s to "AND"s and flipping less-than and greater-than checks:

```
private boolean entitiesOverlap(Entity entity1, Entity entity2) {
    return entity1.getX() < entity2.getXBound()
        && entity1.getXBound() > entity2.getX()
        && entity1.getY() < entity2.getYBound()
        && entity1.getYBound() > entity2.getY();
}
```

Feel free to use this method exactly as is in your term project.

Collision Handling

Once we've detected that a pair of entities has collided, we need to determine what to do based on the specific entity 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

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. Both tanks should move along that axis in opposite directions, and for the smoothest and most predictable result, we'll have both tanks move an equal distance.

To determine that axis and direction 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

Luckily, handling collision between shells is much, much simpler. They both simply 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.

One thing you'll need to be careful about here -- the moment after a tank shoots a shell, it is very likely that your collision detection will determine that that shell immediately collides with the tank shooting it. This is in part due to the bounding box approach, where the boxes for the tank and the shell overlap even if they don't appear to graphically. Obviously, we don't want tanks to immediately shoot themselves whenever they attempt to shoot, so we'll need to have a workaround. One suggested approach -- each shell should remember the ID of the tank that shot it. If it is colliding with that tank, you can simply ignore the collision.

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 (rather than 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 two extra features to the game. I've provided a number of acceptable options; if you have your own ideas, feel free to incorporate them into the game! I encourage you to reach out if you want to know whether your ideas are sufficient for credit.

If you implement more than two features, you can get up to 10 points of extra credit!

Complex AI Logic

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 demoed in class (the TurretAiTank/CushionAiTank), but feel free to build upon that idea! Some examples:

- AI tank that "leads" the player tank by pointing and shooting shells not directly at but some distance in front of the player tank.
- 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

Add power up items to the game that the player tank can pick up! I've provided image assets 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.
- You can add different **Shell** subclasses that correspond to power-ups. For example, a Shell that is capable of turning and tracks the nearest tank would effectively be a homing shell. If the player tank picks up the power up, it could be updated to shoot homing shells instead.

Better Collision Detection

The collision detection algorithm described above is a brute-force approach -- meaning, 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.).

A potential improvement would be to implement a more efficient collision detection algorithm. One simple example would be to split the entire game world into a grid of boxes, say 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 100 by 100 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

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:

```
if (entity1 instanceof Tank && entity2 instanceof Tank) {
  // ...
} else if (entity1 instanceof Tank && entity2 instanceof Shell) {
  // ...
} else if (...) {
  // ...
}
```

The logic here can quickly get out of hand, and is 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.

Feel free to reach out for guidance on this one if you decide to tackle it -- it's one of my favorite design aspects of the entire project, but it's also somewhat advanced and tricky to get correct.

Improve Polish and Game Flow

Even with everything else implemented, the Tank Game may still be lacking in certain flourishes that would make it feel like a "real" game. There are a lot of visual and flow improvements you can make to accomplish this:

- **Animations**: When shells hit a target, or when tanks and walls are destroyed, an explosion animation can go a long way towards improving the visual polish of the game. I've uploaded some files (**Animation.java**, **AnimationResource.java**, updated **RunGameView.java**, and several images for "resources") to handle loading animations for you -- try using these to add explosion animations when shells, tanks, and walls are destroyed!
- **Game UI**: Add some game information indicators on screen during gameplay, such as tank health, current power-ups, or perhaps a score indicator.
- **Better Transitions**: When the game changes screens, some smoother transitions can help the game not feel as abrupt like a demo. Try adding, for example, a 3, 2, 1... countdown when the game starts, or a message indicating if the player won or lost at the end before the end menu screen pops up.
- **Sound**: You'll need to dig into Java Swing code to learn how to incorporate sound into the program. Being able to add sound effects can drastically improve the general polish of the game.
- **Pause Menu**: Allow the player to pause mid-game, with options to restart or quit. Implementing this will also require you to dig into Java Swing code.

Other ideas are welcome! In order to get full credit for this as an extra feature, you will need to implement **at least 3 improvements**.