Tanks 2D - Design Manual

By: Sebastian Ascoli, Jonathan Basom, Minh Bui, Steven Lovine

Introduction

Tanks 2D is an interactive video game based where the user controls a tank and must shoot other tanks without being shot first in order to win the game. The game consists of 2 different game modes: single player and two player. In the single player mode, the user has to beat 10 different levels in order to beat the game. In each level, the user battles against computer controlled tanks. In the two player mode, two players fight against each other and the player who dies first loses. If the 2 players collide with each other, or for some reason die at the same time, then it is a tie.

The game was written in Java, and it is based on Slick 2D (a library designed to make 2D video games based on the popular library LWJGL). The game also uses Jamepad, an open source library that provides support for popular game controllers such as Xbox controller and Playstation controllers. This game was developed using the SCRUM framework.

User stories

- **Completed:**
  - As a single player, I want a tank so that I can move around.
  - As a single player, I want walls so that a tank cannot go through them.
  - As a single player, I want to be able to shoot cannons so that I can attack other tanks
  - As a single player, I want an enemy tank so that I can attack it
  - As a single player, I want the enemy tank to try to shoot me to make a competitive game
  - As a player, I want an obstacle course/maze to make the game more challenging
  - As a single player, I want the enemy to move so that it can try to attack me
  - As a player, I want a main menu where I can choose a game mode or change the settings
  - As a player, I want a settings menu so that I can change the settings
  - As a player, I want a settings option so that I can change the display size settings
  - As a single player, I want the tank that is shot first to lose so that a winner is determined
  - As a player, I want a single player survival mode so that I can see how many games in a row I can win
  - As a player, I want a settings option so that I can change the difficulty level of the game (how fast / how many bullets)
  - As a player, I want music in the main menu

    - As a player, I want a local two player mode so that I can play against friends
    - As a player, I want to be able to play with an Xbox/PS4 Controller so that I can have controller options
    - As a player, I want to have a number of lives so that I can continue playing if I die
    - As a player, I want the enemy tank to have more randomness so that it is easier for the player
    - As a player, I want a pause button so that I can stop the game and resume
    - As a player, I want a settings option so that I can choose whether or not to view the FPS
    - As a player, I want to be able to control the main menu with an Xbox controller
    - As a player, I want a back button so that I can exit the single player or two player mode whenever I want
    - As a single player, I want a label with the current level so that I know where I am at in the game
    - As a user, I want the icon of the game to be a tank
    - As a single player, I want to be able to have shield power-up to make the game more interesting
    - As a single player, I want a settings option so that I can enable / disable shields
- Partially Completed:
    - As a player, I do not want tanks to be able to run over each other
- Incomplete:
    - As a player, I want a best out of three format for the two player game mode
    - As a player, I want a leaderboard so that I can keep track of my high scores in single player survival mode
    - As a single player. I want a next button in between levels so that the next level only starts when I am ready

<u>Object Oriented Design</u>

Game Pieces:

While the design evolved and improved throughout the course of the project, there were a few classes from the beginning that we knew the system would definitely need. For instance, the system obviously needed tanks. However, we decided that we did not want just one type of tank, but instead, we were going to require at least two types of tanks: player tanks and enemy tanks. Therefore, we concluded that we could make an abstract Tank class and have the player and enemy tank classes inherit from it. As Figure 1 below shows, the all tanks would have similar characteristics regardless of their type, such as being able to fire bullets, knowing its current angle and position, and knowing if it is still alive or not. However, there were some important differences between player tanks and enemy tanks. Figure 2 shows that enemy tanks

must be able to determine where the player tank is, if the path is clear to shoot, and what angle they must rotate to in order to be aiming at the player tank. These responsibilities are not included with the player tank because the user must control and determine this information.

Abstract **Tank**	PlayerTank, EnemyTank
• Knows its dimensions • Knows its current angle and position • Knows its speed & angular speed • Knows if it is currently shooting • Knows the maximum number of bullets it can shoot • Knows its previous location • Knows if it is currently alive • Knows if it is intersecting another object • Knows how to move in all 4 directions • Knows how to shoot • Knows how to die	• Bullet

Figure 1 (above) CRC card for the Tank class

PlayerTank	Tank
• Knows position • Knows maximum number of bullets • Knows speed • Knows dimensions	• Bullet

Abstract **EnemyTank**	Tank SimpleStaticEnemy, MovingEnemy
• Knows dimensions • Knows its speeds • Knows the maximum number of bullets it can shoot • Has a desired angle to shoot • Knows its current angle • Finds the location of the player • Determines if path is clear to shoot the player • Decides which direction to rotate • Aims at the player • Shoots the player	• Tank • Bullet

Figure 2 (right) CRC cards for the child PlayerTank and EnemyTank classes

However, as the project progressed, we gathered new user stories such as the desire for two types of enemy tanks: static and moving tanks. Therefore, we decided to make the EnemyTank class abstract and extend it to two concrete classes: SimpleStaticEnemy and MovingEnemyTank. This can be viewed in the basic UML diagram below.

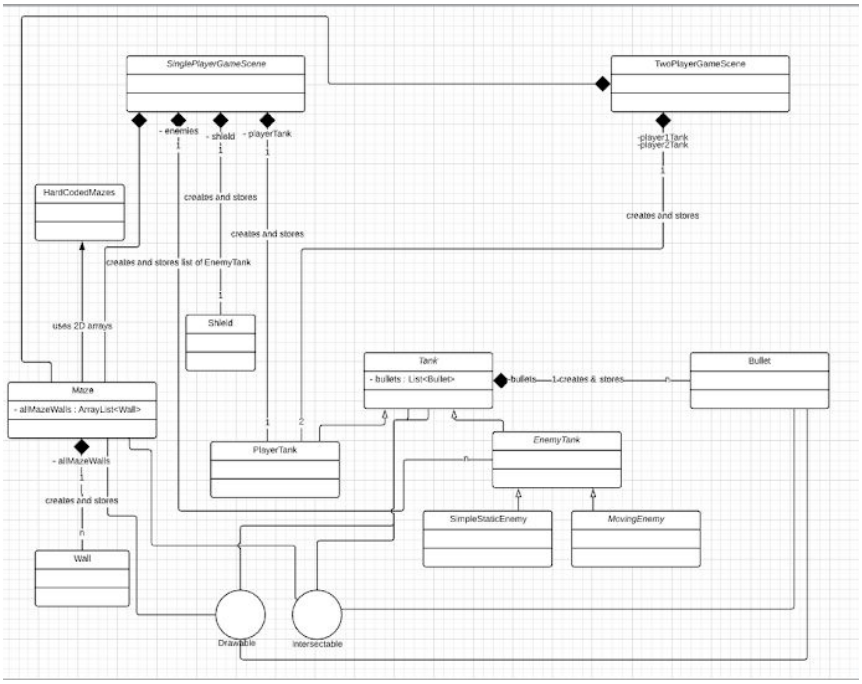

Figure 3 UML Diagram for the pieces of the game

The tanks were just one part of the many classes that were put together in order to form the game. For instance, each tank creates a list of Bullet objects as shown above, and all games need a Maze object for the tanks to traverse. Many of these game pieces shared some common behaviors such as determining if another object intersected it or drawing itself to be rendered to the screen, but each class needed to implement them a bit differently. Therefore, we decided to create two interfaces for the game pieces: Drawable and Intersectable. Drawable allowed the object to be rendered to the screen while Intersectable allowed the object to determine if another object had converged with it. This is quite important for handling collisions between tanks, bullets, and mazes. Figure 4 gives a clearer view of how these classes are related by the two common interfaces.

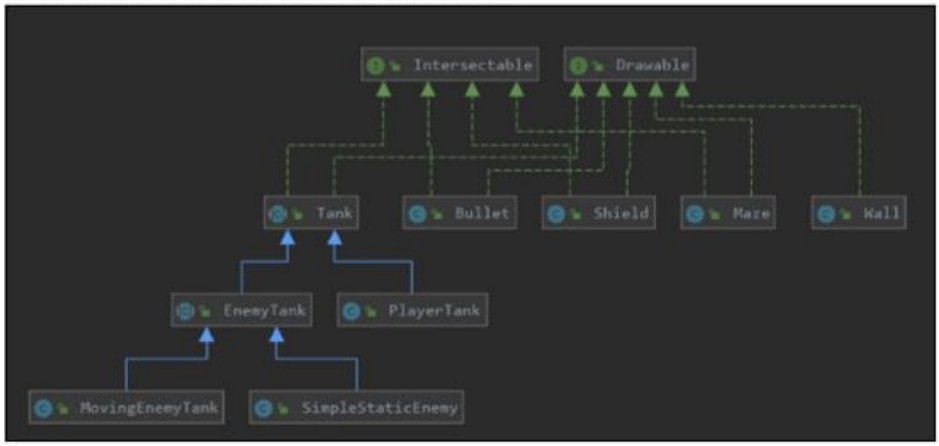

Figure 4 UML of Game Pieces Related by Interfaces

Wall	
• Has dimensions • Has coordinates • Has wall material	• Maze

Maze	
• Has dimensions • Has walls • Has wall material • Determines if an object is intersecting with it	• HardcodedMazes • Wall • GameScene

Figure 4 Wall and Maze CRC Cards

The Maze class is another important part of the gamePieces package that we initially knew that the system would need. Each Maze object contains a 2D array that represents the maze with 1s and 0s. These arrays are stored in the HardcodedMazes class. In order to actually visualize the maze, each Maze object contains a list of Wall objects. As seen above, each Wall has its own dimensions, coordinates, and material. This determines the position of the specific wall, and the material determine what it will look like. The Maze determines where each Wall is located by using the 2D array. Altogether, the game pieces such as the Tanks, Bullets, and Mazes are assembled to create games. These games are extended from the abstract class GameScene.

Game Scenes:

We figured that the game would need several scenes (the main menu, the two player mode, the different levels for the one player mode, the winning scene, the settings, and the instructions).

Slick 2D requires that you have a class inheriting from BasicGame (an abstract class from Slick2D), and you must override the update method (where all the values needed for the game are updated) and the render method (where the actual graphics are rendered).

To be able to easily manage different scenes we created a class called GameManager, which has an attribute for the current scene that is being played. We also have a class called TheGame, which inherits from BasicGame and is the class run by the program's main method. TheGame class calls the GameManager's current scene in the update and render methods. Every "scene" that the GameManager can contain has to implement the Renderable interface (which contains abstract update and render methods), and therefore the GameManager object continuously calls the update and render method of the current scene. Every scene has access to the GameManager object (as it is passed it in the constructor) which allows us to easily switch to another scene.

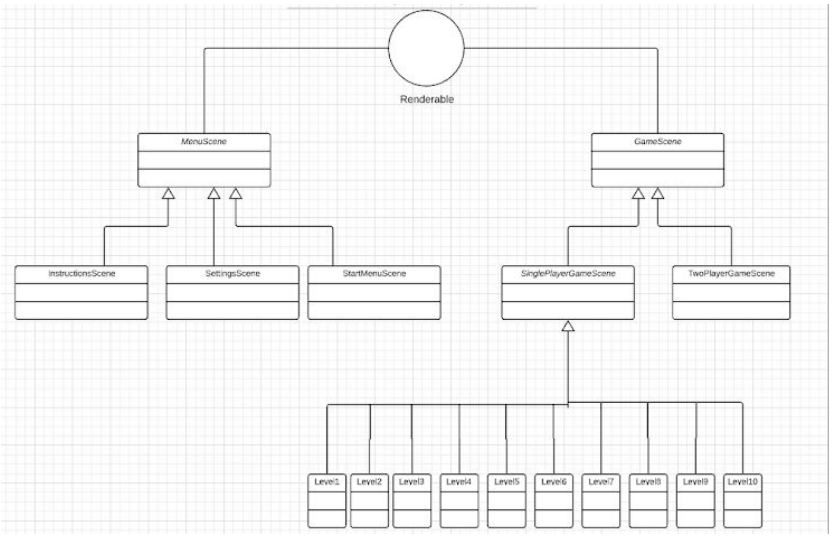

Figure 5 Overview of relationship between scenes

As seen in the UML diagram, the two main types of Renderable scenes are menu scenes and game scenes. The MenuScene class is the parent for scenes such as the StartMenu, SettingsMenu, and InstructionsMenu classes. On the other hand, the GameScene class is the parent of all game classes such as the concrete TwoPlayerGameScene class and the abstract SinglePlayerGameScene class. The SinglePlayerGameScene class is the parent for all of the concrete level classes.

In addition to making switching between different scenes possible (and easy), the GameManager also allows us to store values that would be required by several scenes such as game settings and the ControllerManager object to control video game controllers.

Figure 6 Other responsibilities and interactions for the GameManager class

Summary:

Overall, our initial design changed from the Model View Control design pattern to one that was a bit more compatible with Slick2D. Using Scrum greatly helped as we were able to flexibly adjust and adapt the layout of our system. The Scrum framework allowed us to make changes that would accommodate the needs and wants of the users. As we decided that users wanted more scenes and more levels, we knew that we needed to adjust our design so that we could have a common parent class for all of our levels. This would save us work in the future as we could reuse old code. Therefore, like many other large scale projects, the final design of this game came about through many iterations and adaptations.