CSC 413 Term Project Documentation

Summer 2023

Nhan Nguyen

923100929

CSC413.01

https://github.com/csc413-SFSU-Souza/csc413-tankgame-nhannguyensf

Table of Contents

1 Introduction

1.1 Project Overview (focus of term project)

The project aims to develop a Java-based "Tank Wars Game" application.

Object-Oriented Design Focus: The project is centered around the principles of Object-Oriented Programming (OOP). Concepts like inheritance, encapsulation, polymorphism, and abstraction can be easily identified in the given code.

Real-World Application: By creating a functional game, the project moves beyond theoretical concepts to tangible, real-world application. This facilitates a deeper understanding of how programming concepts come to life in actual products.

Complexity Management: The game involves various components such as sound, user interface elements, game logic, and event handling. By managing and interlinking these components, I learn how to handle the complexity inherent in larger software projects.

Resource Management: The introduction of classes like ResourceManager indicates the importance of efficient resource handling in game development. I need to understand the significance of managing assets, from sounds to images, in a systematic and efficient manner.

Concurrency: The use of separate threads, especially for the game loop, introduces me to the world of concurrency. This is crucial in game development to ensure responsiveness and smooth gameplay, teaching students about the challenges and solutions in concurrent programming.

Event-Driven Programming: The project seems to be event-driven, especially with user interactions and GUI handling. This introduces students to the paradigm where the flow of the program is determined by events, like user actions or system prompts.

In essence, the Tank Wars project offers a holistic approach to studying programming, combining both technical and soft skills. It challenges me to apply theoretical knowledge in a real-world context, fostering deep comprehension and adaptability.

1.2 Introduction of the Tank game (general idea)

The Tank Wars game is a thrilling 2D combat simulation where players maneuver tanks in strategic battles. Set in various landscapes, players must navigate their tanks, defend against enemies, and strategically fire at opponents to claim victory. The objective is simple: outmaneuver and outgun

your adversary to be the last tank standing. Whether battling against computer-driven tanks or challenging friends in multiplayer mode, each encounter is unique, demanding quick reflexes, sharp strategy, and precision shooting. Leveraging advanced Java programming techniques and robust object-oriented design principles, Tank Wars offers an immersive gaming experience. Delve deeper to uncover the cutting-edge technologies and tools that breathe life into this classic warfare game. Some features of the Tank Wars Game:

Dynamic 2D Gameplay: Engage in fast-paced, strategic battles across varied terrains and environments.

Sound Integration: Experience immersive gameplay with integrated background sounds, weapon fire effects, and victory tunes.

Adaptive Game Panels: The game features various panels like Start Menu, Game World, and End Game, ensuring seamless transitions and enhanced user experience.

Customizable Tanks: Personalize your warfare machine with speed, or power-ups.

Real-time Physics: Experience realistic projectile trajectories and tank movements based on actual physics calculations.

Object-Oriented Design: The game's backend is built following solid OOP principles, ensuring modular and maintainable code.

Responsive GUI: The game boasts a user-friendly interface, ensuring smooth navigation and gameplay responsiveness.

Resource Management: Efficiently loads and manages game resources, ensuring optimal performance.

Endgame Scenarios: Encounter unique endgame panels based on your gameplay outcomes, offering varied challenges and scenarios.

More detailed information and technologies used will be provided below.

2 Development Environment

2.1 Version of Java Used

Java 20 (Oracle JDK 20)

2.2 IDE Used

IntelliJ IDEA 2023.2 (Ultimate Edition)

2.3 Special libraries used or special resources

Java Swing: The game uses Swing components (JFrame, JPanel, CardLayout, and JOptionPane) for its graphical user interface. Swing is a Java library that provides tools for building GUI applications.

ResourceManager: appears to be responsible for handling various game resources. It most likely aids in loading and managing assets for the game.

3 How to Build/Import your Project

3.1 Import and build the project:

Follow these steps:

1. Set up the Development Environment:
    i. Ensure you have Java Development Kit (JDK) installed on your computer. You can download the JDK from the official Oracle website.
    ii. Install an Integrated Development Environment (IDE) such as Eclipse, IntelliJ IDEA, or NetBeans. These IDEs provide a user-friendly environment for Java development.
2. Download the Project: Obtain the source code for the Interpreter project. This can be done by downloading the project files from a repository.
3. Open the Project in your IDE:
    i. Launch your chosen IDE and import the project into it. This process may vary slightly depending on the IDE you are using.
    ii. Create a new Java project and configure it to use the existing source code files.
4. Build the Project: Ensure that the project builds successfully without any errors. If there are any compilation errors, review the code and resolve them.
5. Go to File -> Project Structure.
6. Under the Project tab, ensure you have the appropriate JDK selected (JDK 20). If no JDK is available, you can add one by clicking "New..." and selecting the path to your installed JDK.
7. Hover over the "Mark Directory as" option on the "resources" folder to reveal a sub-menu. In the sub-menu, click on "Resources Root". The directory will now be marked as a resources folder.

8. Click the Build -> Build Project option to compile the game.
9. Ensure that the project builds successfully without any errors. If there are any compilation errors, review the code and resolve them.

3.2 How to build the JAR.

- Start IntelliJ IDEA and open Tank Wars game project. Navigate to the 'Project Structure'
- Click on File in the top menu, then choose Project Structure from the dropdown
- Select 'Artifacts': In the Project Structure dialog, choose Artifacts from the left sidebar.
- Add a New Artifact: Click the + button, hover over JAR, and from the submenu, select From modules with dependencies....
- Module and Main Class: A new dialog titled "Create JAR from Modules" will appear.
- Choose the module containing the main class.
- Click the ellipsis (...) next to the Main Class field and select the main class, tankrotationexample.Launcher in this case.
- Finish Setup: Click OK to close the "Create JAR from Modules" dialog. Click Apply and then OK to close the Project Structure dialog.
- Build the Artifact: Go to the Build menu in the top bar and select Build Artifacts....
- Choose the JAR artifact from the list and click the Build button.
- Locate the JAR: Once built, the JAR will be located in the directory that was specified earlier, which is out/artifacts/ by default.
- Testing the JAR: Open a terminal or command prompt, navigate to the JAR's directory, double click to run. Or using the command.

3.3 Commands to run the JAR.

java -jar .\csc413-tankgame-nhannguyensf.jar

4 How to run your game. Rules and controls of the game.

After import/build the project, now we can run the application, follow these steps to run the project:

1. Go to Run -> Edit Configurations.
2. Click the + (plus) button and choose "Java Application" or "Application".
3. Name your configuration.
4. Set the "Main class" to Launcher class.

5. Click the green play button or Run -> Run "YourConfigurationName" to run the game.

5 Assumption Made

During the design and implementation of the project, the following assumptions were made.

Single Platform Focus: The game is primarily designed for desktop platforms, given its usage of Java's Swing framework for GUI elements.

Limited Scalability: The code might be built for a fixed screen size, and might not be designed for resizable game windows or different screen resolutions.

Standalone Execution: The game seems to be intended for standalone execution without reliance on online servers or multiplayer features.

Sound Integration: Given the presence of sound assets in the code (bg, winnerSound, battle), it's assumed that background music and sound effects are integral to the game experience.

Static Resources: The resources (like images, sounds) are likely preloaded and remain unchanged, suggesting there might not be provisions for dynamic content updates or downloadable content.

Card Layout for Navigation: The game relies on Java's CardLayout for navigating between different panels (Start, Game, and End). This implies a linear flow through the game states.

Single Threaded Game Loop: The game loop for updating and redrawing objects runs on its separate thread to ensure GUI responsiveness. It's assumed that this game does not utilize multi-threaded parallel processing for game mechanics.

OOP Centricity: Given the emphasis on OOP in the project instructions, it's assumed that the game emphasizes the use of classes, inheritance, encapsulation, and other OOP principles.

Game End Conditions: Given there's an EndGamePanel, it's assumed the game has defined end conditions, either when a player wins, loses, or possibly other criteria.

Start and End Menus: With Start and End panels, it's assumed that players are greeted with a menu when they start and are shown a summary or option screen when the game ends.

The assumptions provided clarity on the expected behavior and allowed for a more focused development process. It is important to note that these assumptions were specific to the project requirements and may differ if the project were to be expanded or tailored for different use cases.

6 Class Diagram

There is an UML diagram provides a visual representation of the project's class hierarchy, showcasing their relationships and interactions (like inheritance, aggregation, association, or dependencies) with each other.

Please see the picture of UML diagram on **Canvas** or in the ***/documentation*** folder.

7 Class Descriptions

GameObject is an abstract class representing any object that can be displayed and interacted with in the game world. It provides a factory method, newInstance, that constructs specific game objects based on the provided type, such as tanks, walls, power-ups, and more, by using the appropriate resources from ResourceManager. This class enforces the implementation of methods related to collision detection (collides), rendering (drawImage), and getting the bounding rectangle for collision (getHitBox) and activity status (isActive).

PowerUp interface defines a contract for all power-up items within the game. Any class that implements this interface should provide an implementation for the applyPowerUp method. This method dictates how a specific power-up will affect or modify the properties or behaviors of a Tank object when collected.

Health class is a type of GameObject that also implements the PowerUp interface. It represents a health power-up item in the game. The main functionality of the Health object, besides being drawn on the screen and being checked for collisions, is the applyPowerUp method. When a tank collides with it, the tank's health increases (through the tank.increaseHealth() method), and the health item becomes inactive (isActive set to false).

Speed class is a subclass of GameObject and implements the PowerUp interface. It represents a speed power-up item in the game that, when picked up, increases the tank's speed. The core functionality of the Speed object includes being rendered on the screen, collision detection, and the applyPowerUp method. When a tank picks up the speed item, the tank's speed is increased using the tank.increaseSpeed() method, and the speed power-up becomes inactive, indicating it has been consumed.

LivesUp class extends the GameObject superclass and implements the PowerUp interface. It signifies a power-up in the game that, when collected, adds an additional life to the tank. The primary

functions of the LivesUp object encompass being displayed on the screen, handling collisions, and the applyPowerUp method. When the tank acquires this power-up, its life count increases through the tank.increaseLife() method, rendering the LivesUp power-up as inactive, thus marking its consumption.

Tank class in the provided code represents a tank object in a game, complete with attributes like health, speed, position, and methods to handle its movement, shooting, and collisions. This tank can move forwards or backwards, rotate, shoot bullets, respond to power-ups, and collide with various game elements, with visual indicators for health, reloading, and lives.

BotAI class extends the `Tank` class and represents an automated tank in the game which actively pursues and shoots at a player's tank. This bot is governed by AI behaviors to rotate, move, maintain boundaries, and attack the player while displaying its reloading status and managing its bullets.

Bullet class extends the GameObject class and represents a bullet in the game. Each bullet has properties like position, velocity, damage, and an active state. When updated, a bullet moves based on its angle and speed. It can collide with various game objects like walls, breakable walls, and tanks, triggering specific effects such as playing sounds, animations, and causing damage.

Wall class extends the GameObject class, representing an unbreakable wall in the game. Each wall has properties like its position and an associated image. The wall has a hitbox that can detect collisions. While the wall does have a method to handle collisions, it currently doesn't implement any specific behavior upon a collision. When rendered, the wall is drawn on the screen using its image and position.

BreakableWall class, which extends the GameObject class, represents walls in the game that can be destroyed. This class has properties for its position, an image for rendering, and a state indicating if it's still active. Each breakable wall has a hitbox for detecting collisions. The wall's active state can be changed using the setActive method. When rendered, the breakable wall is drawn using its image and position, if it's still active.

ResourceManager class in the tank-themed game efficiently manages the loading and retrieval of game resources like sprites, sounds, and animations using HashMaps. This is a useful resource management utility that decouples resource loading from the main game code, ensuring the game code remains cleaner and more maintainable.

StartMenuPanel class extends `JPanel` to create a start menu for the tank game. This panel displays options like "Start", "Controls", and "Exit" on a background image and also shows the game instructions in a pop-up when the "Controls" button is clicked. The panel's footer also credits the developer.

EndGamePanel class is a JPanel derivative that displays the end-of-game interface for a tank game. Depending on which tank wins, it showcases the respective tank's winning image and offers buttons for "Restart Game", "Main Menu", and "Exit". The winner's tank image is determined by the winnerPlayer variable.

GameConstants class serves as a configuration container for the tank game's various dimensions. It contains constants denoting the width and height of the game world, the screen sizes for gameplay, the start menu, and the end menu. This centralizes important game dimensions for easy access and modification.

TankControl class implements the KeyListener interface to handle keyboard input for controlling a tank's movements and actions.

Animation class represents an animated entity, which is a sequence of images (frames) that change over time to give the illusion of motion or some kind of state change.

Sound class is designed to provide an abstraction over the Clip object from the javax.sound.sampled package. It offers various methods to control playback and volume for sound effects or background music in a game.

GameWorld class represents the core gameplay environment of a tank-themed game, managing both player-controlled and AI-controlled tanks. It facilitates the game's initialization, updates the state of game objects, checks for collisions, and handles rendering through split-screen and a minimap. Key functions include the game loop in the run() method, collision checks in checkCollision(), and graphical rendering in paintComponent().

Launcher class serves as the main orchestrator for the game, initializing and transitioning between the game's various screens using Java Swing's CardLayout. It ensures GUI responsiveness by running the game loop on a separate thread, handles event-driven actions, and manages sound effects based on the game's current state.

8 Self-reflection

Throughout the development of the Interpreter project, I gained valuable insights into various aspects of software development and problem-solving. I had the opportunity to apply and enhance my knowledge of various programming concepts and technologies. Here are some key reflections on the project:

a. Programming Fundamentals: The project allowed me to reinforce my understanding of fundamental programming concepts such as variables, data types, control structures, and functions.
b. Iterative Development: The project taught me the value of iterative development. Building small modules, testing them individually, and then integrating them into the main codebase ensured that I could isolate and rectify issues with ease.
c. Object-Oriented Design: The project emphasized the use of object-oriented programming principles. It helped me comprehend the significance of encapsulation, inheritance, and polymorphism in building modular and extensible software systems. OOP helped in organizing the codebase and promoting code reusability.
d. Importance of Planning: Before diving into coding, I realized the significance of spending time in the planning phase. Mapping out the logic, understanding the intricacies of the interpreter's working, and establishing a clear vision of the desired outcomes were instrumental in steering the project in the right direction.
e. Continuous Learning: The dynamic nature of the project meant that I was constantly on my toes, ready to adapt and learn. Whether it was understanding a new algorithm, debugging technique, or a design pattern, the project was a testament to the ever-evolving nature of software development.
f. Challenges and Overcoming Them: I faced numerous challenges, especially when dealing with intricate parsing and execution nuances. Tackling these obstacles fostered resilience and resourcefulness. Each problem pushed me to seek solutions, whether through research, peer discussions, or trial and error.
g. Collaboration and Feedback: If applicable, working with peers and seeking feedback was enlightening. Diverse perspectives offered fresh takes on problems and opened doors to innovative solutions.

In conclusion, the Term project was a holistic learning experience. It was a melding of theory and practice, challenges and solutions, individual effort and collaborative synergy. Looking back, I feel a sense of accomplishment and gratitude for the insights and skills I've acquired throughout the semester.

9 Project Conclusion/Results

The Tank Wars game project embarked with the primary objective of implementing a 2D game in Java that emphasizes good Object-Oriented Programming (OOP) practices. As I draw to the culmination of this journey, I reflect upon the outcomes, achievements, and learning moments of this endeavor.

Technical Accomplishment: The project's successful execution resulted in a fully functional Tank Wars game that closely adhered to the guidelines and specifications provided. This accomplishment is underscored by the fact that the game offers a seamless user experience, characterized by responsive controls, vibrant graphics, and engaging gameplay mechanics.

OOP Practices: The central tenet of this project was practicing good OOP. Throughout the development, strict adherence to principles like encapsulation, inheritance, polymorphism, and abstraction was maintained. This approach not only streamlined the development process but also resulted in a modular and scalable codebase.

Learning and Challenges: The journey wasn't devoid of hurdles. From grappling with collision detection logic to ensuring smooth animation transitions, various challenges tested the team's mettle. These challenges, however, acted as catalysts for growth, pushing me to innovate and learn.

Collaboration and Teamwork: The project underscored the significance of teamwork. Whether it was peer programming to solve a particularly stubborn bug or class sessions to ideate gameplay features, collaboration was central to my success.

Feedback and Iteration: Continuous testing and gathering feedback played an indispensable role in shaping the final product. Each feedback cycle provided insights that led to refinements, ensuring the game was not only functional but also enjoyable.

Resource Utilization: Leveraging the ResourceManager and other specialized libraries expedited the development process and enhanced the game's overall quality. These tools and resources, combined with custom code, resulted in a balanced and optimized game environment.

Future Prospects: While the current version of Tank Wars stands as a testament to my efforts, there's always room for enhancement. Ideas for additional features, gameplay modes, and improved graphics are already on the horizon, indicating the potential for future iterations of the game.

In conclusion, the Tank Wars game project was more than just a coding exercise; it was a holistic experience that combined technical prowess with creativity, problem-solving, and collaboration. The final game stands as a testament to my dedication, perseverance, and passion for game development. I hope players enjoy battling it out in my virtual tank arena as much as I enjoyed bringing it to life.