# Special Forces vs Robots – OOP2 Project

## Students Details

**Name:** [Musa Abu Alia] **ID:** [208020974] - **Name:** [Ellen Habash] **ID:** [210002721]

## General Description

This project implements a tower defense game called "Special Forces vs Robots" using C++ with SFML graphics library and Box2D physics engine. The game features strategic unit placement, wave-based robot attacks, economy management, and physics-based combat mechanics. Players must defend their base by strategically placing different types of squad members (Heavy Gunner, Sniper, Shield Bearer) to combat incoming waves of robots with varying abilities.

## Design (Architecture)

**Core Architecture**

The project follows a component-based entity system with clear separation of concerns:

**Game Core Objects:**

- **Game**: Main game loop controller managing states and core systems
- **GameManager**: Central game logic coordinator handling economy, health, and game state
- **GameObject/MovingObject/StaticObject**: Base entity hierarchy providing common functionality
- **Robot**: Moving entities with AI behavior and collision detection
- **SquadMember**: Static defensive units with attack capabilities and targeting systems
- **Projectile**: Physics-based ammunition system with trajectory calculations

**State Management:**

- **StateMachine**: Manages game state transitions and rendering stack
- **States (Menu, Play, Pause, Victory, GameOver, Settings)**: Individual game screens with specific responsibilities

**Managers & Systems:**

- **RobotManager**: Handles robot spawning, AI updates, and lifecycle management
- **SquadMemberManager**: Manages unit placement, targeting, and grid-based positioning

- **ProjectileManager**: Controls ammunition firing, collision detection, and physics
- **WaveManager**: Orchestrates enemy waves with dynamic difficulty scaling
- **AudioManager**: Centralized sound and music management
- **ResourceManager**: Asset loading and memory management

**Physics Integration:**

- **PhysicsWorld**: Box2D integration for realistic collision detection
- **PhysicsContactListener**: Handles collision events between different entity types
- **CollisionCategories**: Defines interaction rules between game objects

**Design Patterns Used:**

- **Factory Pattern**: RobotFactory, SquadMemberFactory, ProjectileFactory for object creation
- **Singleton Pattern**: Managers for global system access
- **State Pattern**: Game state management
- **Observer Pattern**: Event system for decoupled communication
- **Command Pattern**: Undo/redo functionality for unit placement and bomb deployment

## File List

**Core System Files**

- **main.cpp**: Entry point and exception handling
- **Game.cpp/h**: Main game loop and window management
- **GameManager.cpp/h**: Central game logic and economy system
- **GameObject.cpp/h**: Base entity class with physics integration
- **MovingObject.cpp/h**: Base for entities with physics movement
- **StaticObject.cpp/h**: Base for grid-based static entities
- **Constants.h**: Game constants, enums, and configuration values
- **Timer.cpp/h**: Enhanced timing system with callbacks

**Entity Implementation**

- **Robot.cpp/h**: Base robot class with AI and combat systems
- **BasicRobot.cpp/h**: Simple melee robot with basic AI

- **FireRobot.cpp/h**: Ranged robot with bullet shooting capabilities
- **StealthRobot.cpp/h**: Fast robot with special movement patterns
- **SquadMember.cpp/h**: Base defensive unit class
- **HeavyGunnerMember.cpp/h**: Standard ranged unit with rapid fire
- **SniperMember.cpp/h**: Long-range high-damage precision unit
- **ShieldBearerMember.cpp/h**: Defensive unit with blocking abilities

**Projectile & Explosive System**

- **Projectile.cpp/h**: Base projectile class with physics trajectory
- **Bullet.cpp/h**: Standard squad ammunition
- **SniperBullet.cpp/h**: High-damage precision ammunition
- **RobotBullet.cpp/h**: Enemy projectiles with squad member targeting
- **Bomb.cpp/h**: Explosive devices with area-of-effect damage and timer-based detonation

**Collectibles & Economy**

- **Collectible.cpp/h**: Base collectible class with lifetime management
- **Coin.cpp/h**: Currency drops from defeated robots
- **HealthPack.cpp/h**: Health restoration items

**Factory Pattern Implementation**

- **RobotFactory.cpp/h**: Creates robots with configuration loading
- **SquadMemberFactory.cpp/h**: Creates squad units with cost calculations
- **ProjectileFactory.cpp/h**: Creates projectiles with physics setup
- **CollectibleFactory.cpp/h**: Creates collectibles with drop rate management

**Manager Systems**

- **RobotManager.cpp/h**: Robot lifecycle and AI coordination
- **SquadMemberManager.cpp/h**: Unit placement and targeting system
- **ProjectileManager.cpp/h**: Ammunition physics and collision handling
- **WaveManager.cpp/h**: Dynamic wave generation with countdown system
- **AudioManager.cpp/h**: Sound effects and music management
- **ResourceManager.cpp/h**: Asset loading and memory optimization
- **SettingsManager.cpp/h**: Configuration persistence and graphics settings

- **EntityManager.cpp/h**: Template-based entity container system

**State Management**

- **StateMachine.cpp/h**: State transition and rendering management
- **MenuState.cpp/h**: Main menu with animated UI elements
- **PlayState.cpp/h**: Core gameplay state with level progression
- **PauseState.cpp/h**: Game pause functionality
- **VictoryState.cpp/h**: Victory screen with Box2D confetti physics
- **GameOverState.cpp/h**: Defeat screen with particle effects
- **SettingsState.cpp/h**: Audio and graphics configuration
- **SplashState.cpp/h**: Loading screen with background asset loading

**UI Components**

- **HUD.cpp/h**: In-game interface with health bars and resource display
- **Button.cpp/h**: Enhanced button class with hover animations
- **GridRenderer.cpp/h**: Visual grid system for unit placement
- **Slider.cpp/h**: Audio volume controls

**Physics Integration**

- **PhysicsWorld.cpp/h**: Box2D world management and stepping
- **PhysicsContactListener.cpp/h**: Collision event handling
- **PhysicsUtils.h**: Coordinate conversion utilities
- **CollisionCategories.h**: Physics collision filtering system

**Animation & Graphics**

- **AnimationSystem.cpp/h**: Sprite animation management with callbacks
- **AnimationComponent.cpp/h**: Entity-specific animation control
- **Animation.cpp/h**: Frame-based animation sequences

**Command Pattern**

- **CommandManager.cpp/h**: Undo/redo system for strategic gameplay
- **PlaceUnitCommand.cpp/h**: Unit placement with rollback capability
- **PlaceBombCommand.cpp/h**: Explosive placement command
- **ICommand.h**: Command pattern interface

**Utility Systems**

- **ConfigLoader.cpp/h**: INI-based configuration file parsing
- **EventSystem.cpp/h**: Type-safe event publishing and subscription
- **DynamicWaveGenerator.cpp/h**: Procedural wave composition

## Key Data Structures

### Primary Containers

- **std::vector<std::unique_ptr<T>>**: Entity storage with automatic memory management
- **std::unordered_map**: Fast asset lookup in ResourceManager and configuration systems
- **std::array<std::array<T, N>, M>**: Fixed-size grid for unit placement optimization
- **std::queue**: Event processing in EventSystem
- **std::stack**: State management and command history for undo/redo

### Game-Specific Structures

- **WaveComposition**: Dynamic wave generation with robot distribution and spawn ordering
- **AnimationComponent**: Frame-based animation with callback system
- **Timer**: Enhanced timing system with progress callbacks and looping
- **Physics Integration**: Box2D body management with SFML coordinate conversion

### Template Usage

- **EntityManager<T>**: Generic entity container with type-safe operations
- **EventSystem**: Template-based type-safe event handling
- **std::unique_ptr/std::shared_ptr**: RAII memory management throughout the project

## Notable Algorithms

### AI and Pathfinding

- **Robot AI State Machine**: Behavior switching between moving, attacking, and special abilities
- **Squad Member Targeting**: Priority-based target selection (closest, strongest, weakest)
- **Dynamic Wave Generation**: Procedural difficulty scaling based on level progression

**Physics and Mathematics**

- **Projectile Trajectory Calculation**: Physics-based ballistic computation with Box2D integration
- **Collision Detection Optimization**: Spatial partitioning through Box2D's broad-phase collision
- **Grid-to-World Coordinate Conversion**: Precise mathematical mapping for unit placement

**Game Systems**

- **Level-Based Economy Scaling**: Mathematical progression for starting resources
- **Animation Frame Interpolation**: Smooth sprite animation with timing calculations
- **Audio Volume Mixing**: Multi-channel audio management with category-based control

**Data Processing**

- **Configuration File Parsing**: INI format parsing with error handling and validation
- **Resource Loading Pipeline**: Asynchronous asset loading with progress tracking
- **Event Queue Processing**: Type-erased event handling with template specialization

## Known Bugs

No known bugs at the moment.

## Additional Notes

**External Libraries**

- **SFML 2.6**: Graphics, audio, and window management
- **Box2D**: Physics simulation and collision detection
- **Standard Template Library**: Extensive use of STL containers and algorithms

**Special Features**

- **Command Pattern Implementation**: Full undo/redo system for strategic unit placement and bomb deployment
- **Explosive Combat System**: Area-of-effect bombs with timer-based detonation and splash damage
- **Physics-Based Combat**: Realistic projectile trajectories and collision detection

- **Dynamic Difficulty Scaling**: Procedural wave generation that adapts to player progress
- **Configuration System**: External file-based game balancing without recompilation
- **Multi-State Architecture**: Clean separation between menu, gameplay, and settings
- **Level Progression System**: Unlockable units and escalating challenge curves
- **Enhanced UI**: Animated buttons, smooth transitions, and visual feedback systems

**Design Patterns Utilized**

The project demonstrates extensive use of object-oriented design patterns including Factory for object creation, Singleton for manager access, State for game flow, Observer for event handling, and Command for user actions, showcasing advanced C++ programming techniques.

**Video Demo**

A gameplay demonstration video has been integrated into the game itself and can be accessed by pressing the "More" button in the main menu, providing an interactive tutorial experience.

**Development Approach**

The project emphasizes modularity, extensibility, and maintainability through clear separation of concerns, consistent coding standards, and comprehensive configuration systems that allow for easy game balancing and feature expansion.