Orbital Defense

1. Project Overview

Orbital Defense is a space-themed tower defense game where players strategically place defensive structures to protect their planetary base from alien invaders.

2. Project Review

This project builds upon concepts from traditional tower defense games like "Bloons Tower Defense" and "Plants vs. Zombies," but incorporates orbital mechanics and space themes to create a unique experience. After studying these existing games, I've identified the following improvements:

1. ***Circular Defense System**: Unlike grid-based tower defense games, Orbital Defense implements a 360-degree approach where enemies come from all directions.*
2. ***Orbital Placement Mechanics**: Instead of placing defenses on fixed grid positions, players place them in circular orbits around the planet.*
3. ***Resource Management System**: Players balance between offensive capabilities (turrets) and economic growth (resource collectors).*
4. ***Comprehensive Statistics**: The game features detailed statistical tracking and visualization to help players analyze their performance.*
5. ***Dynamic Difficulty**: Enemy waves adapt to player performance, creating a personalized challenge level.*

3. Programming Development

3.1 Game Concept

In Orbital Defense, players command the defensive systems of a planet under alien attack. The game focuses on strategic placement

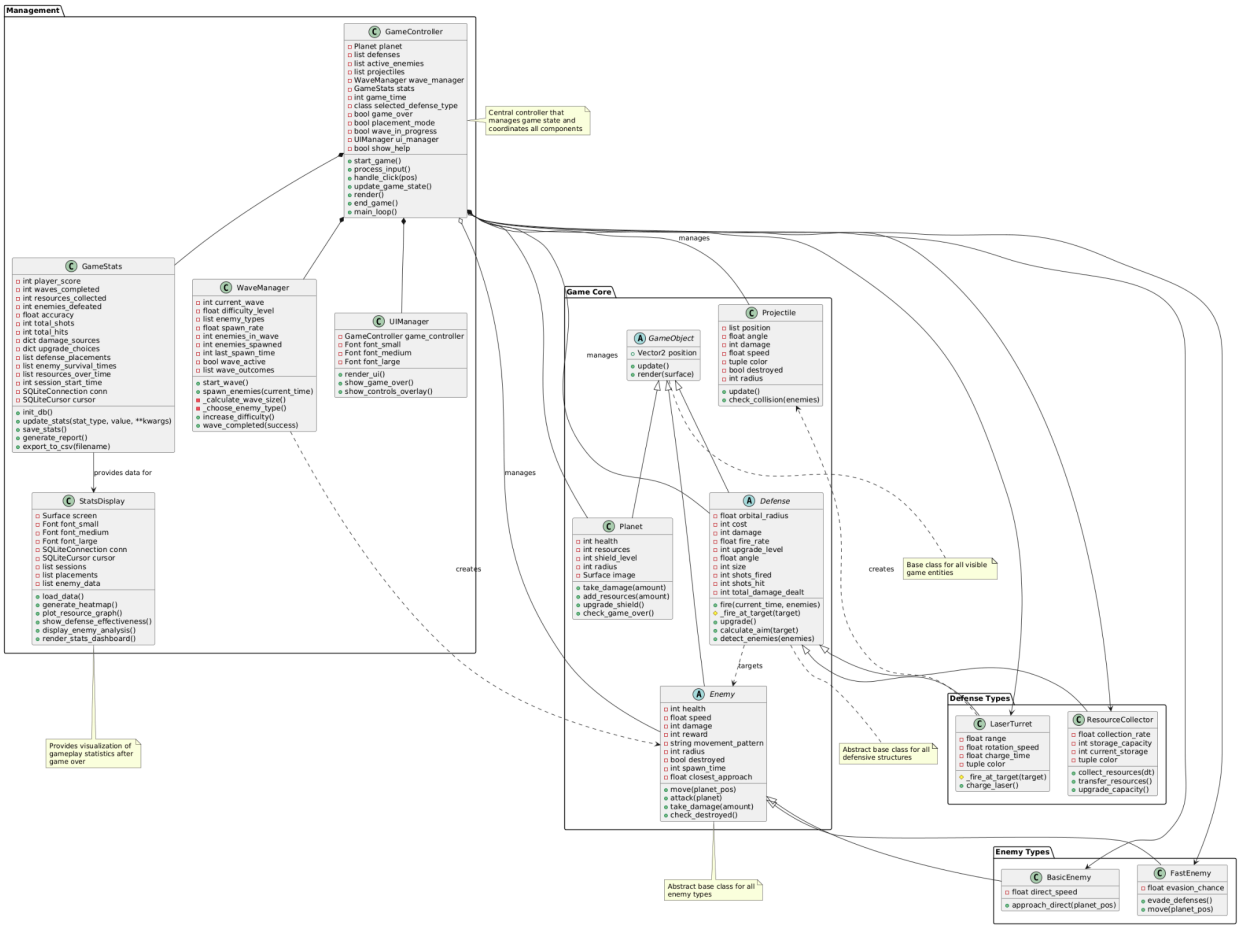
of defenses and resource management across progressive waves of increasingly difficult enemies.

Key Mechanics:

- *Place defensive structures in concentric orbital rings around the central planet*
- *Build and upgrade different types of defenses (laser turrets, resource collectors)*
- *Defend against various enemy types with different movement patterns and abilities*
- *Collect and manage resources to fund defensive expansion*
- *Survive increasingly difficult waves of alien attackers*

3.2 Object-Oriented Programming Implementation

The game implements a object-oriented architecture with inheritance hierarchies and specialized classes:



Class Descriptions:

1. GameObject: Base class for all entities in the game. Provides position and rendering functionality.
    - Attributes: position
    - Methods: update(), render()
2. Planet: The central planet that players must defend.
    - Attributes: health, resources, shield_level, radius
    - Methods: take_damage(), add_resources(), upgrade_shield(), check_game_over()
3. Defense: Abstract base class for all defensive structures.
    - Attributes: orbital_radius, cost, damage, fire_rate, upgrade_level
    - Methods: fire(), upgrade(), calculate_aim(), detect_enemies()

4. Enemy: Abstract base class for all enemy types.
    - Attributes: health, speed, damage, reward, movement_pattern
    - Methods: move(), attack(), take_damage(), check_destroyed()
5. WaveManager: Controls the spawning and progression of enemy waves.
    - Attributes: current_wave, difficulty_level, enemy_types, spawn_rate
    - Methods: spawn_wave(), increase_difficulty(), calculate_next_wave()
6. GameStats: Tracks and records game statistics.
    - Attributes: player_score, resources_collected, enemies_defeated, accuracy
    - Methods: update_stats(), save_stats(), load_stats(), generate_report()
7. UIManager: Handles the user interface and rendering of UI elements.
    - Attributes: game_controller, fonts
    - Methods: render_ui(), show_game_over(), show_controls_overlay()
8. GameController: Central controller for game logic and state management.
    - Attributes: planet, defenses, active_enemies, projectiles, wave_manager
    - Methods: process_input(), update_game_state(), handle_click(), render()

3.3 Algorithms Involved

The game implements several key algorithms:

1. **Orbital Placement Algorithm**:
    - Calculates valid orbital positions based on radial distance and angular separation
    - Ensures proper spacing between defensive structures

    - Uses polar coordinate mathematics to position objects in circular orbits
2. **Enemy Pathfinding**:
    - Basic enemies use direct vector movement toward the planet
    - Advanced enemies implement evasive maneuvers using randomized angular adjustments
    - Fast enemies use pattern recognition to identify and avoid heavily defended areas
3. **Target Priority System**:
    - Implements a weighted decision algorithm for turrets to select optimal targets
    - Considers factors such as enemy health, distance, and type
    - Uses proximity calculations and threat assessment
4. **Dynamic Difficulty Adjustment**:
    - Uses statistical analysis of player performance to adjust enemy wave composition
    - Implements a feedback loop to maintain appropriate challenge level
    - Scales enemy health, speed, and spawn rate based on performance metrics
5. **Data Collection and Analysis Algorithm**:
    - Event-driven data collection during gameplay
    - Real-time aggregation of performance metrics
    - Post-game analysis using statistical methods and visualization algorithms

4. Statistical Data (Prop Stats)

4.1 Data Features

Feature	*Why is it good to have this data? What can it be used for*	*How will you obtain 50 values of this feature data?*	*Which variable (and which class) will you collect this from?*	*How will you display this feature data?*
Player Accuracy	*Helps evaluate player skill and defense effectiveness. Can be used to balance weapon systems and provide feedback to players on their shooting efficiency*	*Each game session records accuracy data. With multiple shots per game across several sessions, we'll easily exceed 50 data points.*	`accuracy`, `total_shots`, and `total_hits` *variables in the* `GameStats` *class.*	***Graph**: Line chart showing accuracy trend over multiple game sessions. Also displayed as a percentage in game summary statistics.*
Defense Placement Patterns	*Reveals player economy management strategy. Shows if players prioritize early resource collection or immediate defense.*	*Each defense placement records orbital radius and angle. With players typically placing 15-30 defenses per game across sessions, we'll collect well over 50 data points.*	`defense_placements` *list in the* `GameStats` *class, which stores position, type, and orbital_radius from the* `Defense` *class.*	***Graph**: Heatmap visualization showing placement frequency across orbital positions. Will use a circular heatmap with color intensity indicating placement frequency.*
Enemy Survival Time	*Identifies difficulty spikes and helps tune the progression curve. Shows where players commonly fail and helps*	*Each enemy spawned has its survival time tracked from creation until destruction. A typical game generates*	`spawn_time` *variable in the* `Enemy` *class, calculated and stored in* `enemy_survival_times` *list in*	***Table**: Statistical summary showing average, minimum, maximum, and standard deviation of*

	balance challenges.	*50+ enemies.*	`GameStats` *when* `Enemy.check_destroyed()` *is called.*	*survival times by enemy type.*
Resource Collection Rate	*Measures the cost-effectiveness and damage output of different defense types. Helps balance defense options and identify under/overpowered types.*	*The game tracks resource collection in 5-second intervals throughout gameplay. With typical games lasting 5+ minutes, we'll collect many data points per session.*	`collection_rate` *variable in the* `ResourceCollector` *class and* `resources_over_time` *list in the* `GameStats` *class.*	***Graph**: Bar chart showing resources collected per game session. Also calculates efficiency metrics (resources per minute) in summary statistics.*
	Helps understand player engagement and identify if games are too short or too long. Reveals pacing issues.	*Each wave completion or failure is recorded. Players progress through numerous waves per game, providing data across different difficulty levels.*	`wave_outcomes` *list in the* `WaveManager` *class, tracked by the* `GameStats.update_stats("wave_completed")`.	***Table and Graph**: Table showing completion rates by wave number. Also displayed as a line graph showing percentage of players who complete each wave number.*
Defense Type Effectiveness	*Each instance of damage to the planet is recorded with its source and amount. Multiple enemies*	*For each defense placed, the game tracks total damage dealt and enemies defeated.*	`total_damage_dealt` *and* `shots_hit` *variables in the* `Defense` *class, tracked*	***Graph**: Comparative bar chart showing damage per resource spent across defense*

	attacking throughout a game provides robust data.	*With multiple defenses per game, each contributing dozens of damage instances, we'll collect comprehensive data.*	*by the* `GameStats` *class.*	*types.*
Play Session Duration	*Helps understand player engagement and identify if games are too short or too long. Reveals pacing issues.*	*Each complete game session time is recorded. Multiple play sessions will provide sufficient data points.*	`session_start_time` *and* `session_end_time` *variables in the* `GameController` *class, calculated and stored by the* `GameStats` *class.*	***Table**: Statistical summary showing average, median, minimum, and maximum session durations.*
Damage Sources	*Shows which enemy types cause the most damage and at what stages players are most vulnerable. Helps tune enemy damage values and identify balance issues.*	*Each instance of damage to the planet is recorded with its source and amount. Multiple enemies attacking throughout a game provides robust data.*	`damage_sources` *dictionary in the* `GameStats` *class, updated when* `Planet.take_damage()` *is called.*	***Graph**: Stacked bar chart showing damage taken from different sources across waves.*

4.2 Data Recording Method

The game implements a comprehensive data recording system using multiple storage methods:

1. ***SQLite Database**: Primary storage format using a relational database with three main tables:*
    - `game_sessions`*: Stores overall session data including date, duration, score, and aggregate metrics*
    - `defense_placements`*: Records detailed data about each defense placement including type, position, and effectiveness*
    - `enemy_data`*: Tracks enemy-specific information including survival times and damage dealt*
2. ***CSV Export**: Secondary backup format for easy analysis in external tools:*
    - *Automatically generates CSV files for each game session*
    - *Includes summary statistics and detailed performance metrics*
    - *Named with timestamp for easy identification*
3. ***In-Memory Tracking**: During gameplay, data is stored in memory using:*
    - *Lists for sequential data (*`enemy_survival_times`*,* `defense_placements`*)*
    - *Dictionaries for aggregated data (*`damage_sources`*,* `upgrade_choices`*)*
    - *Variables for real-time metrics (*`accuracy`*,* `resources_collected`*)*

The data collection happens through an event-driven approach, where the `GameStats.update_stats()` method is called whenever relevant game events occur (enemy defeated, defense placed) This ensures comprehensive and accurate data capture without impacting gameplay performance

4.3 Data Analysis Report

The game includes a statistical dashboard that provides comprehensive analysis through multiple visualization methods:

***Table Analysis**:*

- ***Statistical Summaries**: For each tracked feature, calculates mean, median, minimum, maximum, and standard deviation*
- ***Comparative Analysis**: Cross-tabulation of defense types vs. effectiveness*
- ***Wave Performance**: Success rates and completion times by wave number*
- ***Progress Tracking**: Player improvement metrics across multiple sessions*

***Graphical Analysis**:*

- ***Defense Placement Heatmap**: Circular heatmap showing placement frequency and effectiveness*
- ***Resource Collection Bar Chart**: History of resources collected across game sessions*
- ***Enemy Survival Box Plots**: Distribution of survival times by enemy type*
- ***Accuracy Line Graph**: Trend analysis of player accuracy over time*
- ***Damage Source Stacked Bar Chart**: Breakdown of damage sources by wave*

Feature Name	*Graph Objective*	*Graph Type*	*X-axis*	*Y-axis*
Defense Placement Patterns	*Show strategic preferences in defense*	*Heatmap*	*Orbital position (angular)*	*Orbital radius (radial)*

	positioning			
Resource Collection	*Compare resources gathered across game sessions*	*Bar chart*	*Game sessions*	*Resources collected*
Player Accuracy	*Track improvement in targeting over multiple sessions*	*Line graph*	*Game sessions*	*Accuracy percentage*
Enemy Survival Time	*Compare effectiveness against different enemy types*	*Box plot*	*Enemy types*	*Survival time (ms)*
Damage Sources	*Identify most threatening enemy types*	*Stacked bar chart*	*Wave number*	*Damage amount*

5. Project Timeline

Week	Task
1 (10 March)	Proposal submission / Project initiation
2 (17 March)	Full proposal submission
3 (24 March) (25%)	Set up project structure and repositories, implement core GameObject, Planet, and basic rendering
4 (31 March) (50%)	Complete UI system with placement mode Implement GameStats data collection

5 (7 April) (75%)	Implement statistics dashboard with graphs
6 (14 April) (>90%)	Submission week (Draft) and Testing

6. Document version

Version: *4.0*

Date: *30 March 2025*