2025-group-11b

2025 COMSM0166 group 11b

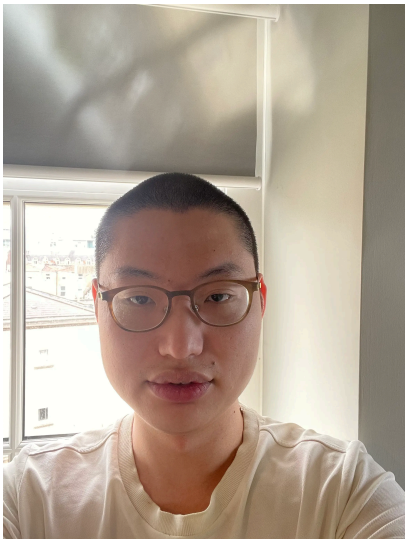
Defend Bristol

Link to your game PLAY HERE

Your game lives in the /docs folder, and is published using Github pages to the link above.

Include a demo video of your game here (you don't have to wait until the end, you can insert a work in progress video)

Your Group



- Group member 1, yangyang, df24465@bristol.ac.uk, role
- Group member 2, Siyuan Chen, gd23774@bristol.ac.uk, developer & product owner & video designer & writer & scrum master

VIDEO

VIDEO LINK : https://youtu.be/Re5hUP_9Xms

✨ 1. Introduction

In designing our game, we set out to create a historically immersive tower defense experience that is both engaging and educational. Our vision was to craft a game that combines strategy , progression , and cultural significance , allowing players to explore the rich history of Bristol while defending the city from various threats across different time periods .

A Unique Twist on Tower Defense

Unlike traditional tower defense games that focus only on mechanics, our game integrates historical storytelling into the experience. Spanning multiple eras, from the medieval period to the high-tech future, players take on the role of a Guardian , defending Bristol from invading armies, pirate raids, and cyber warfare across time.

- **Invading armies**
- **Pirate raids**
- **High-tech cyber warfare**

Get ready to **defend, strategize, and experience history like never before!**

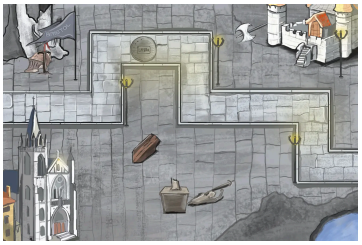


Figure 1
Medieval Map

Figure 2
Age of Exploration Map

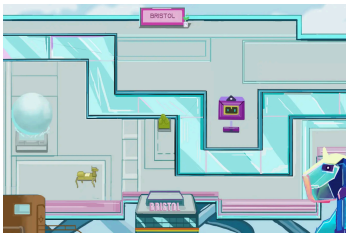
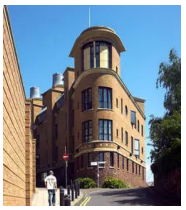


Figure 3
Future Map

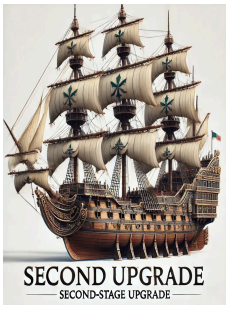
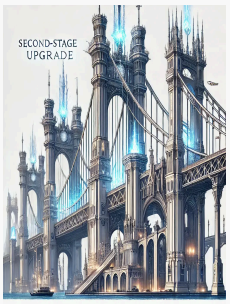
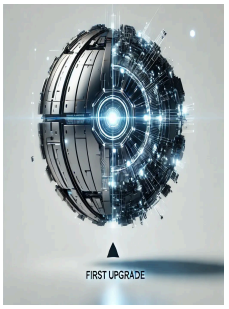
Bringing History to Life

To enhance immersion, we carefully incorporated **real historical landmarks** from Bristol, such as:

Landmark	Image	Landmark	Image
Bristol Cathedral		**SS Great Britain**	
Clifton Suspension Bridge		**St Mary Redcliffe**	
Cabot Tower		**Merchant Venturers Building**	

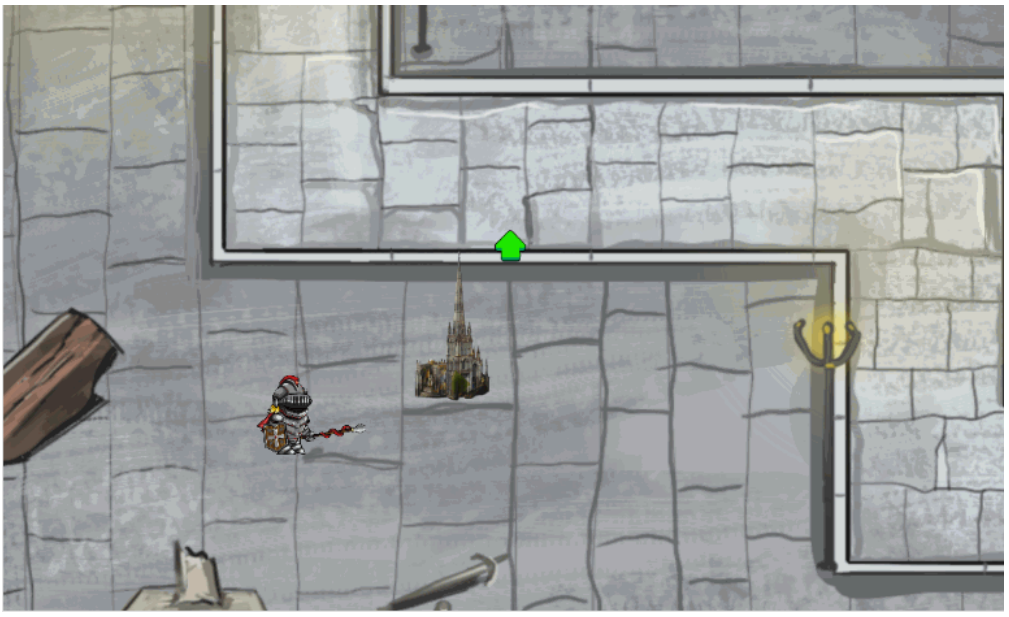
Chemistry Building		**We The Curious**	

Beyond this, we have also implemented a distinctive tower upgrade mode , allowing players to evolve their defenses in historically meaningful ways.

Base Tower	Upgrade 1	Upgrade 2
		
		
		

Knight Mechanic: Absorbing Tower Abilities

To tackle the common lack of player interaction in tower defense games, we introduced a medieval knight who can absorb a tower's ability by staying nearby for 5 seconds . The knight then fights alongside the tower, adding a dynamic and strategic layer as players guide him between different towers to combine powers and defeat enemies.



🎯 2. Requirements

Ideation Process - Game Concept Development

Early Brainstorming & Game Ideas

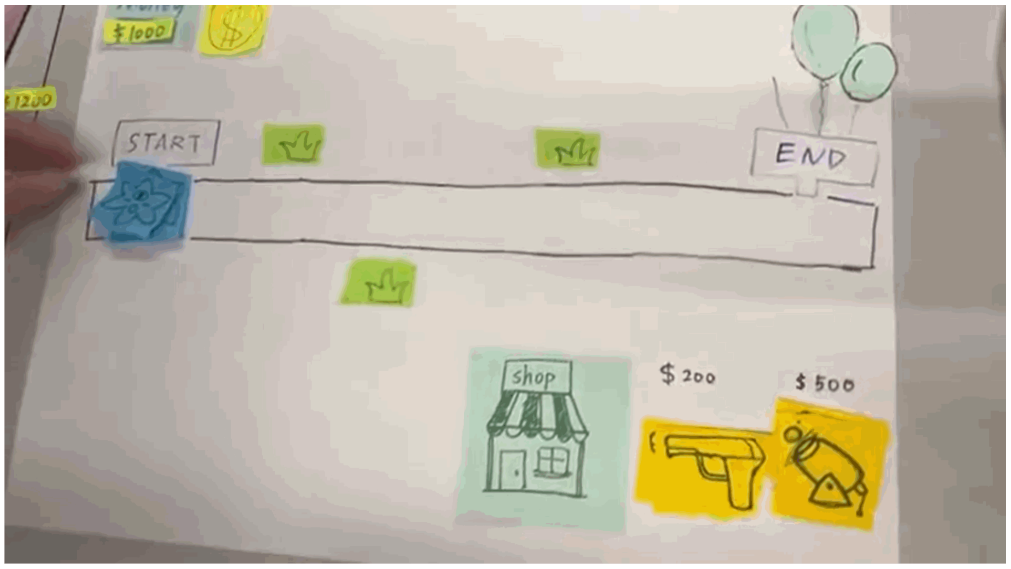
During **Week 2**, our team (**which initially had 6 members**) brainstormed **two potential game concepts**:

1. **Tower Defense Game**
    - A **historical tower defense game**, integrating **different defense strategies across various eras**.
    - Players would be able to **place, upgrade, and manage towers** to defend against invaders from different time periods (e.g., knights , pirates , futuristic AI robots).
2. **Aircraft Shooter Game**
    - A **classic aerial shooter**, where players pilot an aircraft to **engage in air combat and eliminate enemies**.
    - The game would feature multiple historical settings (e.g., **WWI, WWII, modern warfare, and futuristic battles**).

Paper Prototype Development

To better understand and test the game mechanics, we created two paper prototypes for this as part of the early validation process：

- **Tower Defense Game Paper Prototype**



- **Aircraft Shooter Game Paper Prototype**

How We Decided as a Team What to Develop (which initially had 6 members)

- Open Brainstorming Sessions:
  We first conducted open discussions where every team member pitched ideas, focusing on gameplay mechanics, target audiences, and possible historical or educational elements.
- Paper Prototype Feedback:
  We created paper prototypes (see below) for both the Tower Defense Game and the Aircraft Shooter Game. By testing these prototypes in small play sessions, we gathered direct feedback on fun factor, ease of learning, and potential for expansion.
- Feasibility & Resource Assessment:
  Given our team size and the complexity of each concept, we weighed the required development effort (e.g., coding complexity, art assets, AI pathfinding) against our available skills and time constraints.

The Tower Defense concept was deemed more feasible due to clearer scope definition and manageable AI requirements.

- Stakeholder & Audience Considerations:
  We referred to our Onion Model of Stakeholders to anticipate who would be most invested in each idea. We also considered which concept might have broader appeal and educational potential.
- Final Team Vote & Milestone Check:
  After considering feedback, testing the paper prototypes, and reviewing our resources, the team held a final vote. We chose the Tower Defense concept.

Onion Model of Stakeholders

To better visualize the relationship between different stakeholders in our game project, we created an **Onion Model** to represent their roles and levels of involvement.

The model illustrates four concentric layers:

- The innermost layer represents the **Players**, who directly interact with the game system.
- The second layer includes the **Core Development Team** — Designers, Developers, Artists, and QA Testers — who build and test the game.
- The third layer consists of the **Support & Management Team** — Project Managers, Sound Designers, and Community Moderators — who ensure project progress and community engagement.
- The outermost layer features **External Stakeholders** — Marketing Team, Investors, Publishers, and Potential Partners — who promote, fund, and expand the game's reach.

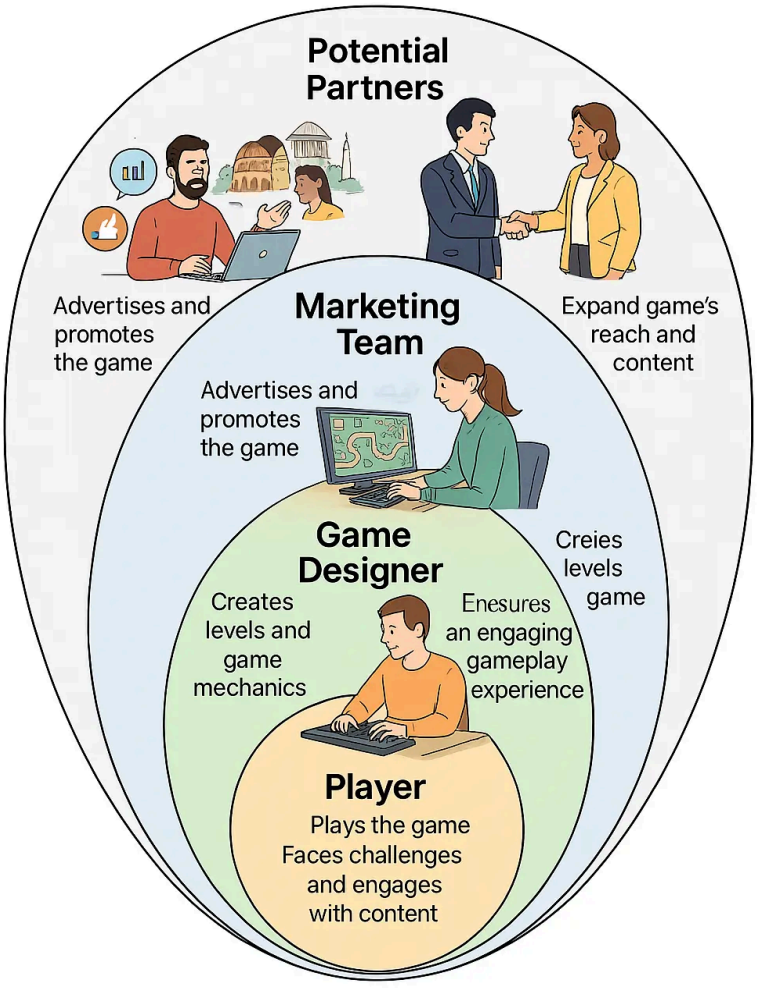


Figure 4

Onion Model

Identify Top-Level User Needs

After analysing the project context and stakeholders' needs, we identified the top-level user requirements (Epics) that would guide our game development:

- Epic 1: Provide engaging gameplay with progressive difficulty.
- Epic 2: Allow players to explore historical periods through gameplay.
- Epic 3: Offer a variety of towers and enemy types to ensure diversity and replayability.
- Epic 4: Optimize game performance and stability across different devices.
- Epic 5: Create a scalable backend and infrastructure to support future updates and multiplayer functionality.

User Stories & Stakeholders

In this phase, we identified the key **stakeholders** whose input and needs would directly influence the game's development. Our goal was to ensure the game appealed to a broad audience—both users and creators. The key stakeholders we focused on were:

- **Players**: The main users who play the game.

- **Game Designers**: Those who design levels and mechanics.
- **Developers**: Programmers implementing the game.
- **QA Testers**: People testing for bugs and balance.
- **Artists**: Designers of game assets (characters, towers, environment).
- **Sound Designers**: Those who create sound effects and music.
- **Project Managers**: Oversee development progress.
- **Marketing Team**: Promote the game to players.
- **Investors/Publishers**: Fund the project.
- **Community Moderators**: Manage player discussions and feedback.

As we refined our ideas, we began developing **user stories** to ensure that our vision aligned with the experiences we wanted to deliver. These stories were crafted in **collaboration** with developers, designers, and artists, incorporating a variety of perspectives to make sure we were covering all aspects of the game.

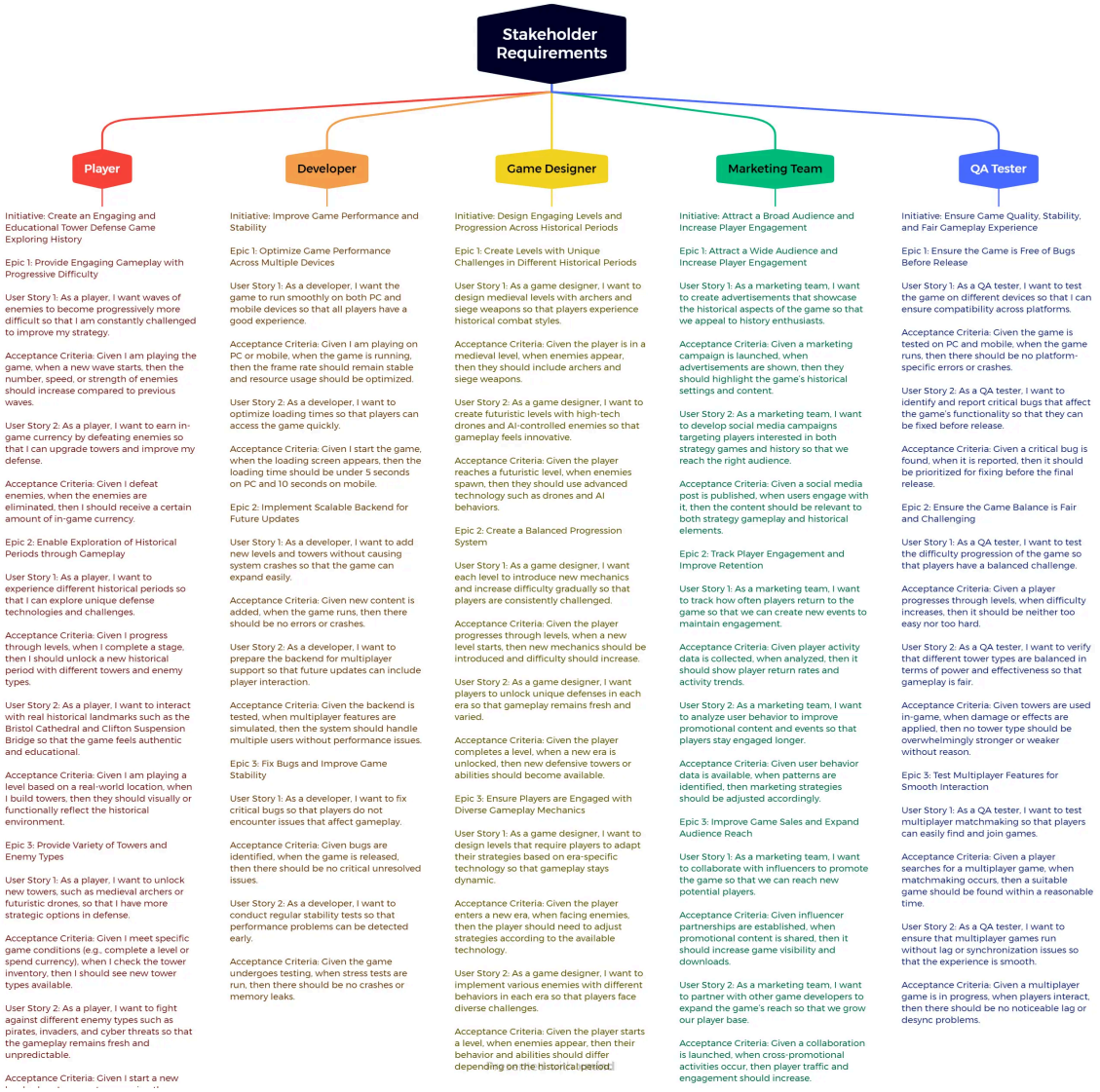


Figure 5

Stakeholder Requirements

Specify Atomic Requirements

To further refine our user needs, we specified atomic requirements based on our user stories. These requirements are clear, testable, and independent to ensure traceability and system verification.

Example Atomic Requirements:

Requirement ID	Requirement Description	Acceptance Criteria
AR-01	The system shall allow the player to place a tower on an empty grid.	Given the player has enough resources, when they select a grid, then the tower should be placed successfully.
AR-02	The system shall allow the player to upgrade a tower to increase its attack damage.	Given the player selects an existing tower, when they choose upgrade, then the tower's attack stats should increase.
AR-03	The system shall spawn enemy waves with increasing difficulty.	Given the player starts a new wave, when the wave spawns, then enemy stats (speed, health, number) should increase.
AR-04	The system shall display earned in-game currency when enemies are defeated.	Given an enemy is defeated, when the player views their resources, then the currency value should update accordingly.

Early Design & Prototyping

Research & Conceptualization

In the early stages of design, we researched traditional tower defense games to understand their mechanics and identify potential areas for innovation. Our key decision was to merge tower defense gameplay with the historical evolution of Bristol 🏛️, offering players the opportunity to enjoy strategic gameplay while also learning about the city's history.

Core Gameplay Design

Once the core game concept was defined, we focused on establishing the key gameplay pillars. For example, tower placement would be strategically important, with different historical periods unlocking unique towers and monsters. We also aimed to incorporate Bristol's historical landmarks as key gameplay elements, enhancing both the strategic depth and the educational value of the game.

Technical Considerations

To ensure the game met these design goals, we considered the system's technical requirements. The game needed to be accessible across multiple platforms (PC, console, and web) , allowing players to engage seamlessly. The user interface was designed to be intuitive, facilitating easy tower placement, upgrades, and resource management.

Prototype Development

With these concepts in mind, we developed an early prototype, testing core mechanics such as grid-based tower placement, monster AI pathfinding, and resource management. These tests validated our ideas and ensured that the game could progress to full-scale development with a solid foundation.

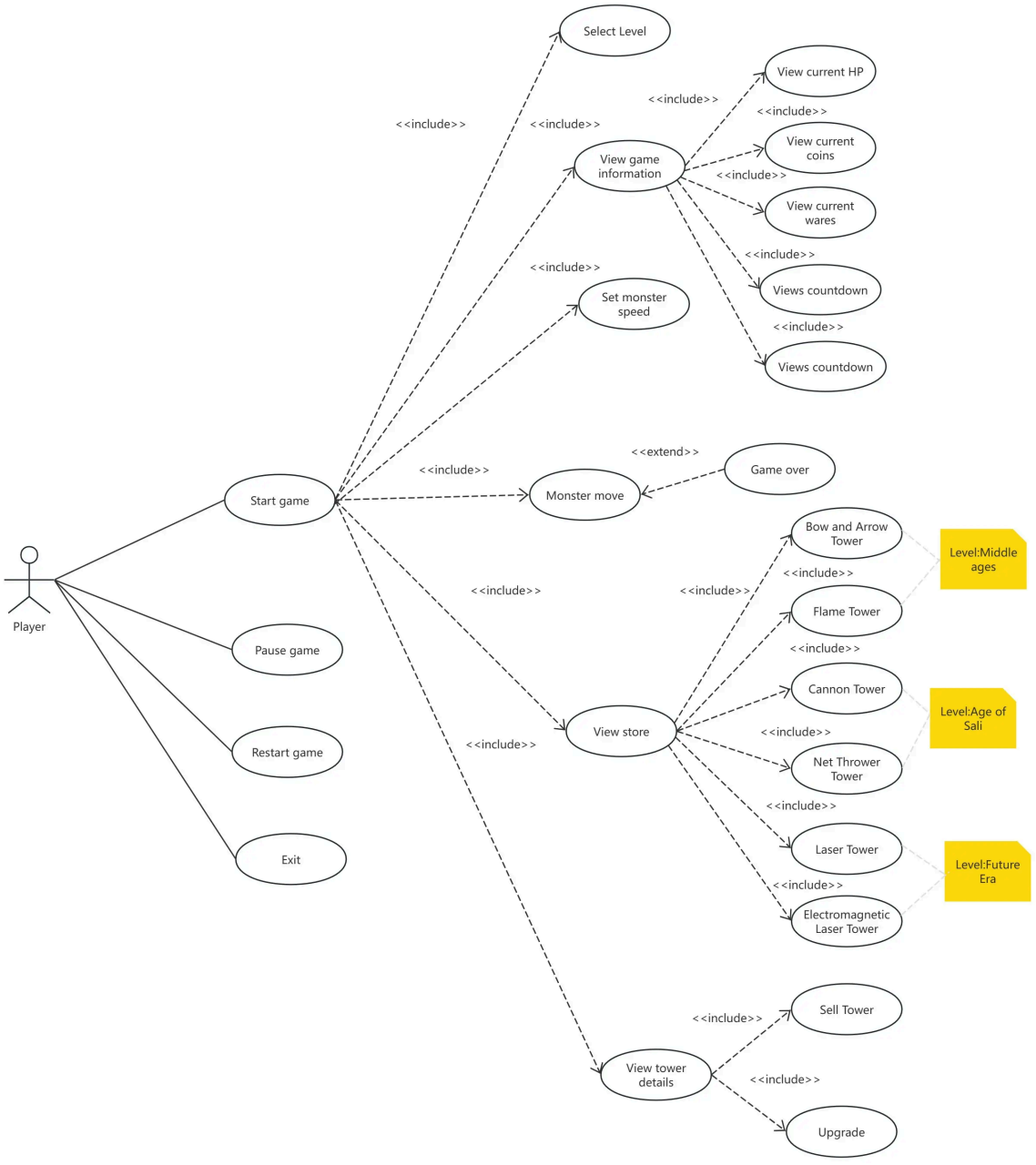
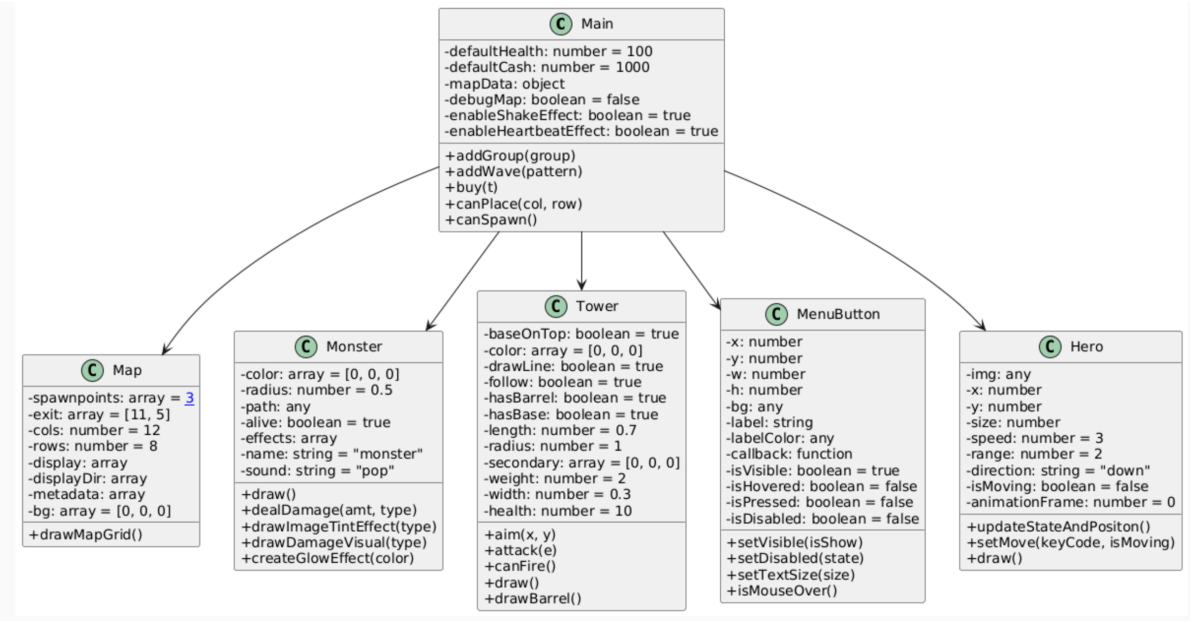


Figure 6

Use case diagrams

3. Design: System Architecture & Diagrams



📦 Class Diagram Explanation – Tower Defense Game

This class diagram illustrates the structure and interactions between major components in a tower defense game system, including monsters, towers, and the game map.

🔷 Class: `main` (Manager Class)

This is a **controller/manager** class for managing entities in the game such as monsters, bullets, towers, and paths.

Attributes:

- `monsters`, `newMonsters`: Arrays managing current and incoming monsters.
- `bullets`, `newBullets`: Arrays for handling bullet instances.
- `towers`, `newTowers`: Tower entities.
- `particle`: Handles particle effects (e.g., explosions).
- `paths`: Movement paths for monsters.
- `wave`, `cash`, `health`: Game state indicators.

Methods:

- `dealDamage(x, y)`, `attack(monster)`: Core game logic for combat.
- `draw()`, `shake()`, `update()`, `move()`: Visual or state update logic.
- `onKilled(monster)`, `ifDie()`: Death handling for monsters.

🏰 Class: `Tower`

Represents a defense tower placed by the player.

Attributes:

- Boolean flags for visuals and logic: `baseOnTop`, `drawLine`, `length`, `radius`, `type`
- Vectors and arrays: `color`, `pos`
- Stats: `cost`, `damageMin`, `damageMax`

Methods:

- `rotate(x,y)`, `attack(monster)`, `onTarget(monster)`: Tower targeting and firing logic.
- `draw()`, `destroy()`, `initialize()`, `sell()`, `resetCd()`: Rendering and lifecycle functions.

👾 Class: `monster` (Entity Class)

This is the **individual monster entity**, separate from the manager class.

Attributes:

- Identity and state: `name`, `health`, `cash`, `speed`, `color`
- Visuals and animation: `frameIndex`, `frameCount`, `animationSpeed`, `facingRight`
- Position: `pos`

Methods:

Same as manager's monster logic:

- `dealDamage(x,y)`, `attack(monster)`, `draw()`, `move()`, `update()`
- `ifDie()`, `onKilled(monster)`, `shake()`

🗺️ Class: `map`

Represents the level layout and environment configuration.

Attributes:

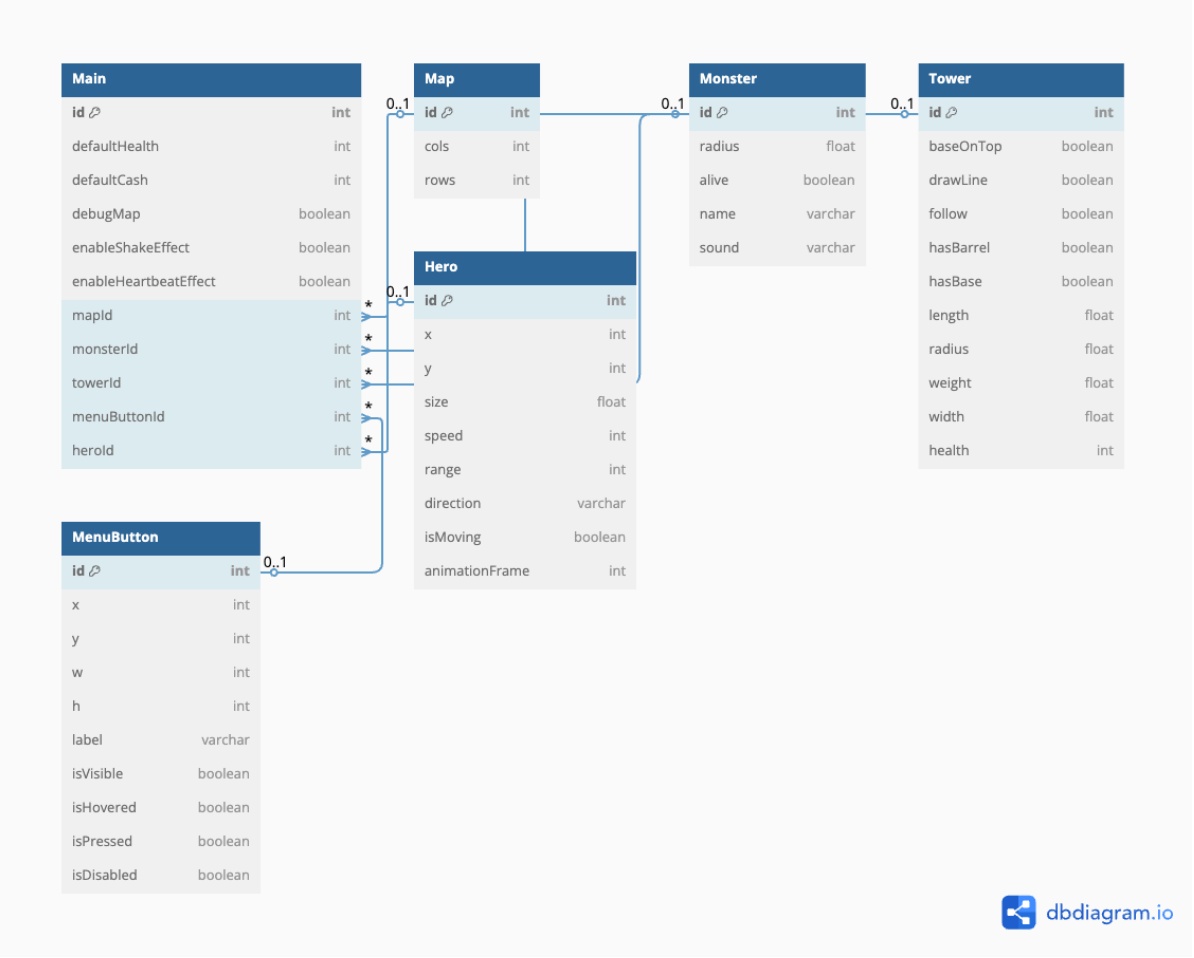
- `customMap`: Full map configuration object.
- `grid`, `paths`, `metadata`, `spawnpoints`, `exitArray`: Layout and logic elements.
- Dimensions: `bgArray`, `cols`, `rows`

🔁 Class Relationships

- The `monster` manager **controls** both `Tower` and individual `monster` entities.
- The `map` is **loaded** and referenced by the `monster` manager.
- Towers **attack** individual `monster` entities.
- The manager class `monster` is distinct from the monster instance class `monster`.

🧠 Notes

- There is a separation between **game state management (controller)** and **entity objects**.
- The system uses arrays to manage dynamic entities like bullets and monsters.
- Inheritance is not shown; composition and control logic dominate.



📘 Entity-Relationship Diagram Description

🧩 Entities

Main

- Central controller of the game.
- Holds game-wide settings and references to other components.
- Attributes:
  - `defaultHealth: number = 100`
  - `defaultCash: number = 1000`
  - `mapData: object`
  - `debugMap: boolean = false`
  - `enableShakeEffect: boolean = true`
  - `enableHeartbeatEffect: boolean = true`
- Methods:

- `+addGroup(group)`
- `+addWave(pattern)`
- `+buy(t)`
- `+canPlace(col, row)`
- `+canSpawn()`

Map

- Represents the game map/grid and its structure.
- Attributes:
  - `spawnpoints: array`
  - `exit: array`
  - `cols: number = 12`
  - `rows: number = 8`
  - `display: array`
  - `displayDir: array`
  - `metadata: array`
  - `bg: array = [0, 0, 0]`
- Methods:
  - `+drawMapGrid()`

Monster

- Represents an enemy/creature in the game.
- Attributes:
  - `color: array = [0, 0, 0]`
  - `radius: number = 0.5`
  - `path: any`
  - `alive: boolean = true`
  - `effects: array`
  - `name: string = "monster"`
  - `sound: string = "pop"`
- Methods:
  - `+draw()`
  - `+dealDamage(amt, type)`
  - `+drawImageTintEffect(type)`
  - `+drawDamageVisual(type)`
  - `+createGlowEffect(color)`

Tower

- Represents a defensive tower in the game.
- Attributes:
  - `baseOnTop: boolean = true`
  - `color: array = [0, 0, 0]`
  - `drawLine: boolean = true`
  - `follow: boolean = true`
  - `hasBarrel: boolean = true`
  - `hasBase: boolean = true`
  - `length: number = 0.7`
  - `radius: number = 1`
  - `secondary: array = [0, 0, 0]`
  - `weight: number = 2`
  - `width: number = 0.3`
  - `health: number = 10`
- Methods:
  - `+aim(x, y)`
  - `+attack(e)`
  - `+canFire()`
  - `+draw()`
  - `+drawBarrel()`

MenuButton

- UI element for user interaction.
- Attributes:
  - `x, y, w, h: number`
  - `bg: any`
  - `label: string`
  - `labelColor: any`
  - `callback: function`
  - `isVisible: boolean = true`
  - `isHovered: boolean = false`
  - `isPressed: boolean = false`
  - `isDisabled: boolean = false`
- Methods:
  - `+setVisible(isShow)`
  - `+setDisabled(state)`

- `+setTextSize(size)`
- `+isMouseOver()`

Hero

- The main controllable player character.
- Attributes:
  - `img: any`
  - `x, y: number`
  - `size: number`
  - `speed: number = 3`
  - `range: number = 2`
  - `direction: string = "down"`
  - `isMoving: boolean = false`
  - `animationFrame: number = 0`
- Methods:
  - `+updateStateAndPosition()`
  - `+setMove(keyCode, isMoving)`
  - `+draw()`

🔗 Relationships

From	To	Relationship Type	Description
Main	Map	1 : 1	One Main has one Map
Main	Monster	1 : *	One Main controls many Monsters
Main	Tower	1 : *	One Main controls many Towers
Main	MenuButton	1 : *	One Main controls many MenuButtons
Main	Hero	1 : 1	One Main controls one Hero

🧩 Tower Defense Game - Sequence Explanation

1. Tower Placement

- The **User** selects and places a tower.
- The **GameLoop** initializes the tower by adding it to the `towers` array.

---

2. Game Loop Updates

The **GameLoop** continuously runs, updating each frame:

- **Monster** moves and updates its state.
- If the monster exits the map, it triggers a **health decrease**.
- **Tower** targets nearby monsters and updates its state.

---

3. Tower Attacks

- The **Tower** chooses the nearest monster as a target.
- A **bullet is spawned** via `bullets.push()`.
- The **Bullet** is updated via `steer()` and `update()` functions.

---

4. Collision & Explosion

- The **GameLoop** checks if the bullet hits the target using `reachedTarget()`.
- If hit:
  - The **bullet explodes** and is **destroyed**.
  - A **visual particle effect** is triggered via `particle.run()`.

---

5. Wave Check

If all monsters are defeated (`noMoreMonster()` returns `true`), the system:

- Checks for **wave completion**.
- Starts the **next wave** by calling `nextWave()`.

4. Implementation

Building a Modular Tower Defense Game with p5.js

Before developing more complex gameplay features—like hero skills or dynamic wave logic—we first designed a modular and scalable architecture for a tower defense game. The game logic was split into multiple dedicated modules: towers.js handled tower mechanics, monsters.js managed enemy behavior, maps.js stored predefined map layouts, main.js served as the core game loop, and ui.js managed interface components and interactions.

Although map data itself is statically defined in maps.js, we implemented dynamic pathfinding and map validation logic through functions like getWalkMap() and getVisitMap(). This allowed the system to simulate flexible terrain behavior and makes it extensible for future features like random map generation or a map editor.

A custom SlidePane class was implemented to provide a tower information panel that can expand or collapse with animation, supports scrollable content, and dynamically updates display data. It also includes interactive elements like upgrade and sell buttons. This layer of UI polish significantly improved the gameplay experience.

Inside main.js, the main game loop manages all subsystems, including monster spawning, tower targeting, projectile and particle systems, and player status such as health and gold. While each module handles its own responsibilities, they coordinate via shared state flags such as toPlace, paused, and toCooldown.

Though this may appear to be a relatively simple tower defense game, we placed strong emphasis on code structure, separation of concerns, and future-proof design. This foundation allows us to easily expand the game with new features such as skill trees, map editors, or new enemy types without rewriting core logic.

Challenges

1. Multi-resolution Support and Complex UI Interaction

One of the most significant challenges we encountered during the development was ensuring that our game could properly support multiple screen resolutions while maintaining the integrity of a complex user interface. The game includes interactive UI elements such as a tower selection panel, a scrollable information bar, and paginated navigation buttons. In the early stages of development, we used hardcoded pixel-based coordinates for positioning UI elements. While this approach worked initially on our target resolution, it quickly became problematic when tested on devices with different screen sizes or aspect ratios. UI components would become misaligned, clipped, or overlapping, severely impacting usability. To address this, we refactored the layout system to rely on relative positioning and scaling, often using percentage-based anchors and layout groups provided by our game engine. We also incorporated a dynamic resolution manager that could adjust element spacing, font size, and margins in real-time. This significantly improved the consistency of the user experience across various devices and resolutions, though it required a considerable redesign of our original UI structure.

1.1 Initial Use of Static HTML and UI Loading Order Issues

The first challenge arose from our initial decision to build the game's interface using static HTML pages. While this approach allowed for rapid prototyping and easier organization of UI components, it introduced a noticeable issue during the game's loading phase. Specifically, we observed that the sidebar UI—containing menus and options—would render on the screen before the main game canvas was fully initialized. This led to a disjointed user experience where players would momentarily see an incomplete or partially loaded interface, which gave the impression of a performance issue or a broken layout.

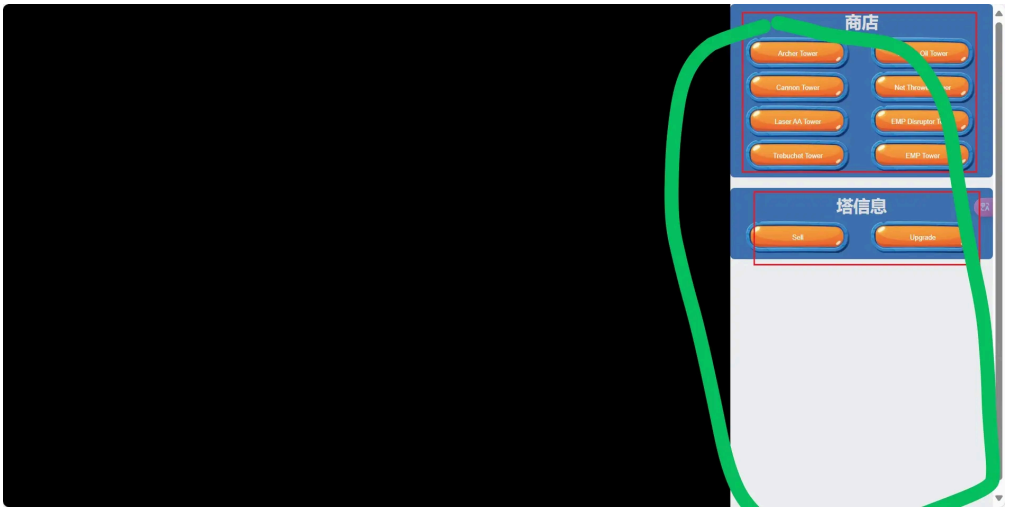


Figure 7

use html instead of p5

To address this issue, I decided to abandon the use of static HTML altogether and refactored all UI pages into a unified p5.js-based interface. By relying solely on p5 for both game logic and UI rendering, we eliminated the premature rendering problem caused by HTML loading separately from the game canvas. This integration ensured that all elements appeared simultaneously and consistently once the canvas was fully initialized.

However, this change introduced a new challenge. Since the entire interface was now rendered through p5, its appearance became highly sensitive to browser-specific scaling behaviors and screen DPI settings. In different browsers or under varying resolution and zoom configurations, the sidebar layout could shift, scale inconsistently, or even overflow the visible canvas area. These discrepancies posed a new threat to the visual stability we had just achieved and required us to implement additional logic to detect screen resolution and zoom level, and dynamically adjust the canvas scale and element positions accordingly.

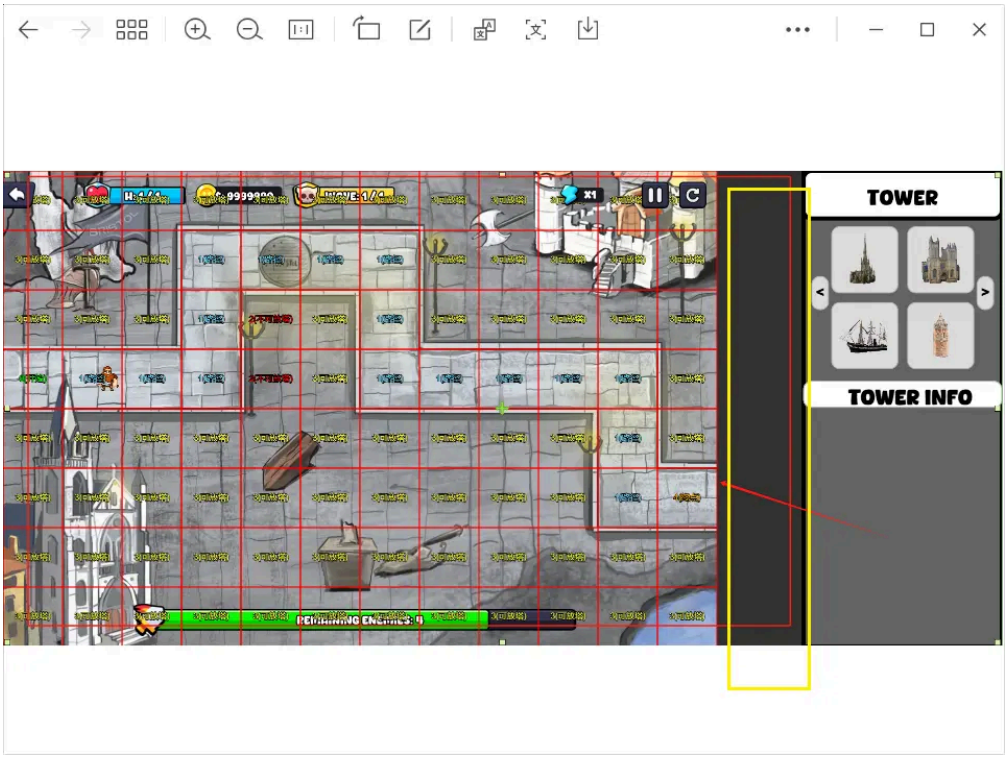


Figure 8

bad attempt

1.2 Final Unified Rendering Solution and Adaptive Scaling Strategy

To ensure a consistent and responsive user experience across a wide range of screen sizes and aspect ratios, we implemented a dynamic resolution adaptation strategy using p5.js. At the core of this approach is the calculation of a scalable tile size (ts), which is derived based on the current window dimensions and the predefined grid size (12 columns by 8 rows). This ensures that all map elements maintain their proportions and remain visually aligned across different devices.

The game dynamically computes the canvas dimensions by multiplying the tile size with the number of columns and rows, and then recenters the canvas on screen using calculated offsets (gameX and gameY). This allows the game view to remain centered and properly scaled regardless of window size. UI components such as buttons, panels, and fonts are also scaled proportionally using a scaling factor derived from the tile size, ensuring a unified appearance.

Furthermore, we utilize the windowResized() function to listen for screen changes in real time. Whenever the window is resized, the canvas is updated, UI elements are re-rendered, and layout parameters are recalculated to maintain consistency. This approach offers a resolution-independent, pixel-perfect layout that avoids layout breaks and maintains usability across devices, including mobile and ultra-wide screens.

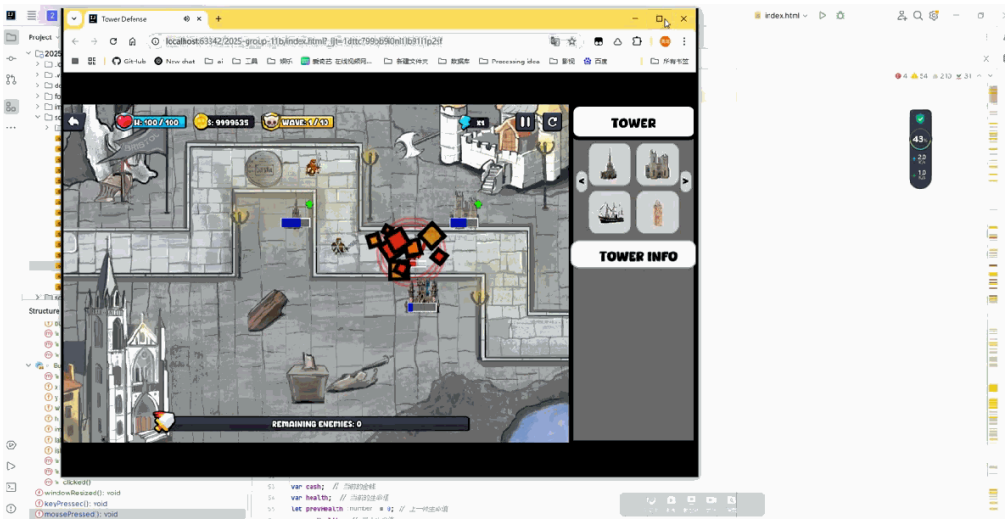


Figure 9

final solution

2. Designing Enemy and Tower Spawning Systems

In the early stages of development, one major challenge we faced was how to effectively manage the spawning of enemies and the placement of towers within the game world. Initially, we adopted a very static approach: enemies and towers were represented by fixed images (sprites) that were directly placed onto the screen using absolute pixel coordinates. While this allowed for a quick visual prototype, it quickly proved insufficient for a scalable tower defense game.

The use of static images meant that each tower and enemy type had to be manually drawn and positioned, which not only limited their interactivity but also introduced alignment issues across different screen resolutions. Additionally, this approach offered no room for animation, dynamic behaviors, or in-game upgrades—all of which are crucial features in a tower defense experience.

We soon realized that to support features like tower upgrades, enemy health tracking, and real-time interactions (e.g., targeting and attacking), we needed to move toward a fully object-oriented architecture. Each tower and enemy would need to be represented as a class instance, with properties such as position, type, health, damage, and cooldown logic. This transition laid the groundwork for creating a robust and flexible system that could scale with the game's increasing complexity.

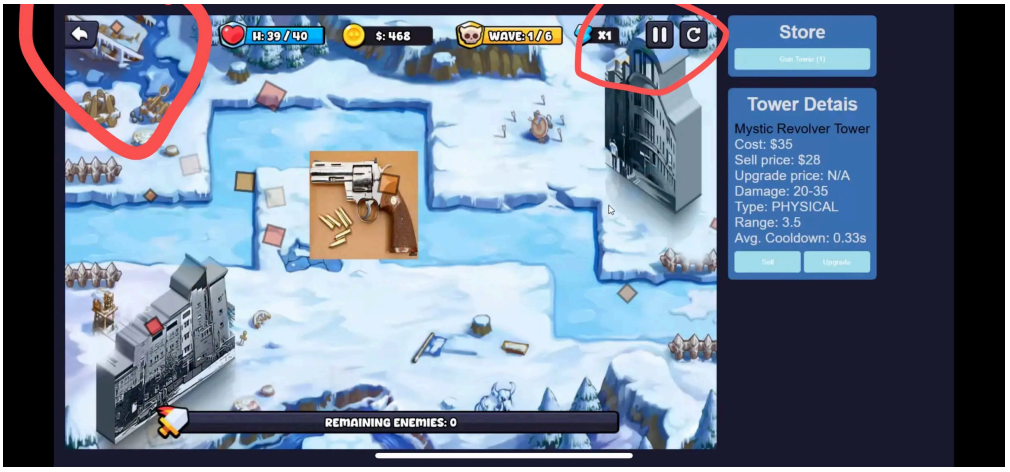


Figure 10

the very beginning

As the game evolved, we moved beyond using static images and began designing enemies as fully encapsulated class-based objects. Each enemy is now an instance of a Monster class, which not only stores its position, health, speed, and pathfinding logic, but also manages animation frames, status effects, damage response, and visual effects such as stun, burn, or freeze.

The transition to this structure enabled us to implement sprite-based animations in a scalable way. Rather than hardcoding image switching, each enemy has internal state properties like frameIndex, frameCount, and animationSpeed, allowing frame-based animation to be updated smoothly in real time. These properties make it possible to visually differentiate enemy types and behavior phases, such as walking, stunned, or dying.

In addition, the Monster class supports rendering various attack effects (e.g., lightning arcs, frost spikes, or glow pulses) and integrates seamlessly with the combat and targeting system. This change drastically improved both gameplay feedback and code maintainability. By centralizing all enemy logic within a single class, we gained a modular structure that's easier to expand with new enemy types, behaviors, or animations in the future.

Figure 11

sprite sheet 1

Figure 12

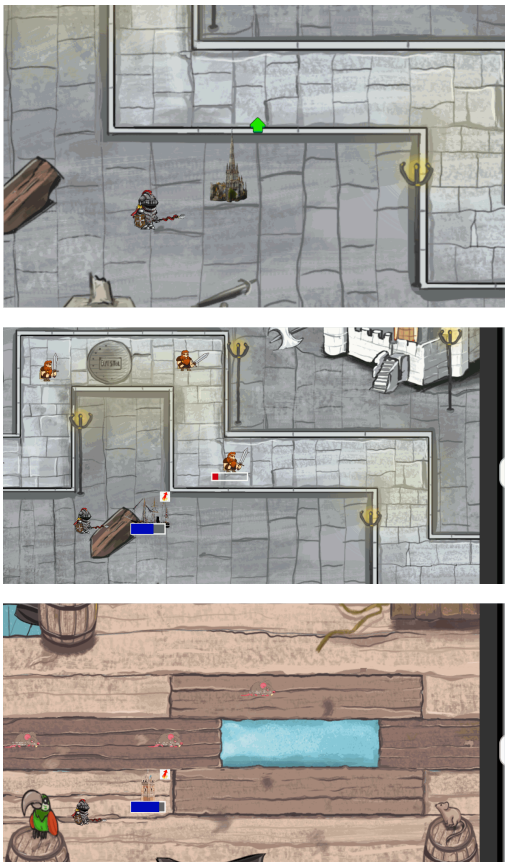
sprite sheet 2

3. Introducing an Interactive Knight System to Overcome Passive Gameplay

Traditional tower defense games often suffer from a lack of interactivity: once towers are placed, the player's role becomes largely passive, limited to watching waves unfold or clicking upgrades. To address this, we introduced a controllable knight system as an active gameplay element.

The knight is implemented as a movable Hero class, which can be directly controlled using keyboard inputs to navigate the battlefield in real time. This character is not just cosmetic. It can actively interact with the environment, detect nearby towers, and temporarily acquire their abilities through proximity-based collision detection. Once empowered, the knight inherits the selected tower's attack type, cooldown, and targeting logic, essentially acting as a mobile tower with dynamic targeting and skill reuse.

This system introduced several design challenges, including collision detection between the knight and towers, cooldown management to prevent ability spamming, and synchronization of animation states with movement direction. We also had to balance the knight's influence so that it would complement the tower mechanics rather than replace them. The result is a more engaging player experience that blends real-time action with classic tower defense strategy.



🎯 5. Evaluation

Why Evaluation Is Important

Conducting thorough evaluations is a crucial step in user-centered game development. By collecting qualitative feedback early and iteratively, we can rapidly identify usability obstacles, refine our mechanics, and ensure that **Defend Bristol** meets real player expectations. It also helps prevent costly redesigns later on, aligning with industry best practices for delivering polished gaming experiences.

Choosing a Qualitative Evaluation Method

To explore how players truly experience our game, we considered two commonly used qualitative approaches:

Method	Definition	Advantages	Limitations
Heuristic Evaluation	Experts inspect the game against established usability principles (Nielsen's 10 Usability Heuristics).	- Fast, low-cost, requires no external users. - Systematically checks known interface issues.	- May miss authentic user reactions. - Relies on expert assumptions, not real behavior.

Method	Definition	Advantages	Limitations
Think Aloud Evaluation	Real users verbalize their thoughts, decisions, and emotional reactions as they interact with the game.	- Provides direct insight into genuine user experiences. - Small samples reveal big issues.	- Depends on participants' willingness to talk. - Some may find it awkward while playing.

Nielsen Norman Group

Jakob's Ten Usability Heuristics

1 Visibility *of* System Status

Designs should **keep users informed** about what is going on, through appropriate, timely feedback.

2 Match between System and the Real World

The design should speak the users' language. Use words, phrases, and concepts **familiar to the user,** rather than internal jargon.

3 User Control *and* Freedom

Users often perform actions by mistake. They **need a clearly marked "emergency exit"** to leave the unwanted state.

4 Consistency *and* Standards

Users should not have to wonder whether different words, situations, or actions mean the same thing. **Follow platform conventions.**

5 Error Prevention

Good error messages are important, but the best designs **prevent problems** from occurring in the first place.

6 Recognition Rather Than Recall

Minimize the user's memory load by making elements, actions, and options visible. Avoid making users remember information.

7 Flexibility *and* Efficiency of Use

Shortcuts — hidden from novice users — may **speed up the interaction** for the expert user.

8 Aesthetic *and* Minimalist Design

Interfaces should not contain information which is irrelevant. Every extra unit of information in an interface **competes** with the relevant units of information.

9 Recognize, Diagnose, *and* Recover from Errors

Error messages should be expressed in **plain language** (no error codes), precisely indicate the problem, and constructively suggest a solution.

10 Help *and* Documentation

It's best if the design **doesn't need** any additional explanation. However, it may be necessary to provide documentation to help users understand how to complete their tasks.

Conclusion:
After weighing these methods, we concluded that observing actual players would yield richer, more actionable feedback. Therefore, we ultimately chose **Think Aloud** as our primary method to gain deeper insights into player behavior and emotional responses.

Think Aloud Evaluation

What Is Think Aloud?

Think Aloud involves inviting participants to share their thoughts, feelings, and decision-making processes aloud as they perform tasks in the game. We observe them in real time, noting points of confusion, frustration, or excitement to understand how our design is interpreted by actual players.

Pros and Cons

Pros	Cons
Rich, Immediate Feedback See exactly where players stumble or succeed.	**Reliance on Participant Disclosure** Shy players may hold back thoughts.
Cost-Effective Requires minimal equipment and small participant groups.	**Less Natural Playing** Continuous talking may break immersion.

1. Planning a Think Aloud Evaluation

We began by clarifying our key questions:

- *Can players figure out how to place and upgrade towers on their own?*
- *Do they understand how the Knight absorbs a tower's ability?*
- *Are there any points of confusion about the interface or objectives?*
- *Can players intuitively grasp each tower's unique function and counter mechanics?*
- *Is the economy system (gold/resources) transparent and balanced?*
- *Does the difficulty curve ease players in without early frustration?*

Based on these, we created tasks covering each core mechanic — starting a new game, placing a tower, upgrading, using the Knight to absorb abilities, and completing a wave.

Each session lasted 20–30 minutes with three participants:

- 2 familiar with tower defense games
- 1 completely new to the genre

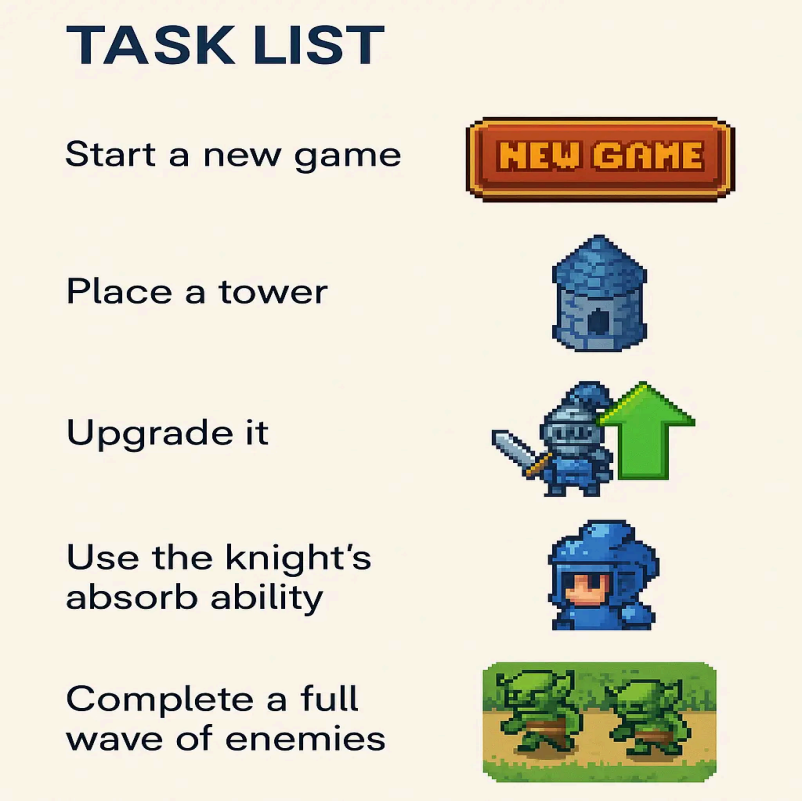


Figure 13

Task List

2. Carrying Out a Think Aloud Evaluation

Roles:

- 1 Facilitator(Siyuan Chen): explains the method and encourages thinking aloud.
- 1 Observers(Yang Yang): take notes on participants' behaviors and comments.

Facilitator instructions example:

> "There are no right or wrong answers — please share any thoughts or feelings you have as you play. If you fall silent for a while, I might gently ask what you're thinking."

Prompt example (if silent too long):

> "Could you talk me through your decision here?"

3. Analyzing a Think Aloud Evaluation

We collected observation notes and categorized feedback into:

- **Helpful Features**
- **Confusing Elements**
- **Suggestions for Improvement**

Summary of Feedback

Category	Positive Feedback	Issues Identified	Suggestions for Improvement
Map Interaction	Historical map design is visually unique.	Difficulty distinguishing walkable areas vs. obstacles.	Highlight walkable areas (e.g., path glow or grid markers).
Tower Management	Grid-based placement feels intuitive.	Unclear upgrade paths; tower types lack visual differentiation.	Add tier icons/colors for upgrades; animate tower abilities on hover.
Enemy Speed Adjustment	Speed tweaks add strategic depth.	Confusion over whether changes apply to all enemies or just new waves.	Add real-time UI feedback (e.g., "Speed +10%: Affects ALL enemies").
UI Controls	Layout is clean and navigable.	Ambiguous button functions (e.g., sell/upgrade).	Relabel buttons (e.g., "Sell Tower - 50% Refund"); add confirmation dialogs.
Tower Function Clarity	Tower variety provides interesting strategic options.	Players struggle to intuit tower abilities and counter mechanics.	Add tooltips with stats (e.g., "Slow Tower: Reduces speed by 30%") and enemy-type hints.
Economy Transparency	Resource system encourages thoughtful decision-making.	Gold/resource flow feels unbalanced; upgrades seem costly for marginal gains.	Show gold-per-wave previews; rebalance upgrade costs to align with power spikes.
Difficulty Pacing	Progressive challenge keeps players engaged.	Early waves are too punishing for new players.	Tune Wave 1-3 enemy health/density; add adaptive hints after failures.

Conclusion and Next Steps

Based on the Think Aloud sessions, we identified key improvements:

- **Improve Map Interaction**
  - Highlight walkable areas to improve map readability and reduce player confusion.
- **Refine Tower Management Feedback**
  - Add visible or audible cues when a tower upgrade is complete.
  - Improve visual differentiation between tower types and upgrade paths.
- **Clarify Enemy Speed Adjustment**

    - Provide clear feedback about the effect of speed-up.
    - Add tooltips or hints when activating speed-up.
- **Refine Wave Alerts**
    - Provide clearer indicators or countdowns before enemy spawning to help players prepare.
- **Clarify Knight Ability Absorption**
    - Incorporate a progress bar or timer to show the absorption duration for better player feedback.
- **Refine UI Controls**
    - Improve button labels for clarity.
    - Add confirmation prompts for critical actions to prevent misclicks.

Choosing a Quantitative Evaluation Method

While our Think Aloud sessions provided rich insights into **Defend Bristol**'s user experience, we also incorporated a standardized quantitative method to validate and measure specific aspects of player perception:

NASA Task Load Index (NASA TLX)

Purpose

Measure the perceived workload that players experience when performing core tasks in the game.

Method

We adopted a within-subjects design: each of the 10 participants played Easy, Mid, and Hard versions of Defend Bristol, then completed the NASA TLX after each session.
This allowed us to perform paired statistical analyses (Wilcoxon signed-rank tests) and reduce the effect of individual differences on workload scores.

The NASA TLX evaluation is computed in two main steps:

1. **Determine Dimension Weights (Pairwise Comparisons):**
   Each participant conducts 15 pairwise comparisons among the six dimensions:
    - **Mental Demand (MD)**
    - **Physical Demand (PD)**
    - **Temporal Demand (TD)**
    - **Performance (Perf)**
    - **Effort (Effort)**
    - **Frustration (Frust)**

   In each comparison, if a dimension is judged as more important for workload, it receives 1 point. As a result, each dimension's weight is an integer between 0 and 5, and the sum of the weights for all six dimensions is always 15.

2. **Rate Each Dimension (0–100 Scale):**
   Each participant then assigns a rating (from 0 to 100, usually in increments of 5) to each dimension.
   Their **Weighted TLX Score** is calculated by multiplying each dimension's rating by its individual weight, summing these products, and dividing by 15:

Individual Participant Weights Table

User	Mental Demand	Physical Demand	Temporal Demand	Performance	Effort	Frustration	Sum
1	5	1	3	2	3	1	15
2	4	2	2	3	2	2	15
3	5	0	4	2	3	1	15
4	3	3	2	3	3	1	15
5	4	1	3	2	4	1	15
6	2	4	3	2	2	2	15
7	3	2	4	1	3	2	15
8	5	1	2	3	2	2	15
9	4	2	2	4	1	2	15
10	3	3	1	3	3	2	15

Ease Difficulty Ratings and Weighted TLX Scores Table

User	Mental Demand	Physical Demand	Temporal Demand	Performance	Effort	Frustration	Weighted Score
1	25	25	33	20	25	10	24.9
2	34	9	32	25	22	22	25.4
3	26	7	32	22	32	6	26.9
4	26	12	34	24	21	17	22.3
5	31	22	26	26	30	9	27.0
6	18	9	22	28	45	23	22.0
7	17	12	24	30	37	6	21.6
8	30	19	12	25	32	8	23.2
9	23	19	13	23	26	6	19.1
10	27	23	32	27	20	19	24.1

Mid Difficulty Raw Ratings & Weighted TLX

User	Mental Demand	Physical Demand	Temporal Demand	Performance	Effort	Frustration	Weighted Score
1	50	22	53	55	53	36	49.1
2	57	34	41	58	65	26	48.9
3	56	32	39	54	46	40	48.1
4	46	22	46	50	58	40	44.0
5	41	25	49	56	65	37	49.7
6	50	32	54	52	60	39	46.1
7	47	30	42	49	45	31	41.0
8	41	32	39	51	57	35	43.5
9	49	31	52	53	60	39	47.5
10	44	39	54	47	55	33	45.0

Hard Difficulty Raw Ratings & Weighted TLX

User	Mental Demand	Physical Demand	Temporal Demand	Performance	Effort	Frustration	Weighted Score
1	90	47	72	80	85	65	79.5
2	74	38	62	78	66	55	64.8
3	71	48	58	82	68	50	67.0
4	65	36	59	75	75	60	62.1
5	84	37	61	83	70	60	70.8
6	67	44	77	77	80	40	62.3
7	82	49	64	85	79	63	69.9
8	69	44	58	82	84	51	68.1
9	77	37	60	81	65	59	67.3
10	88	49	62	79	73	51	68.7

Group Level Mean Weighted TLX by Difficulty

Difficulty	Mean Weighted TLX
Easy	23.6
Mid	46.3
Hard	68.0

Group Level Mean Weighted TLX by Difficulty

Dimension	Easy	Mid	Hard
Mental Demand	6.7	12.2	19.5
Physical Demand	1.9	3.9	5.4
Temporal Demand	4.5	8.0	11.1
Performance	4.1	8.8	13.3
Effort	5.0	9.7	13.0
Frustration	1.4	3.7	5.8

This breakdown reveals a progressive increase in mental, temporal, and effort-related workload with rising difficulty, and a decline in perceived performance, which aligns with expectations in usability studies.

Visualization of Weighted NASA TLX

To better visualize how different task dimensions contributed to perceived workload across difficulty levels, we computed the average weighted contribution of each TLX dimension for Easy, Mid, and Hard conditions.

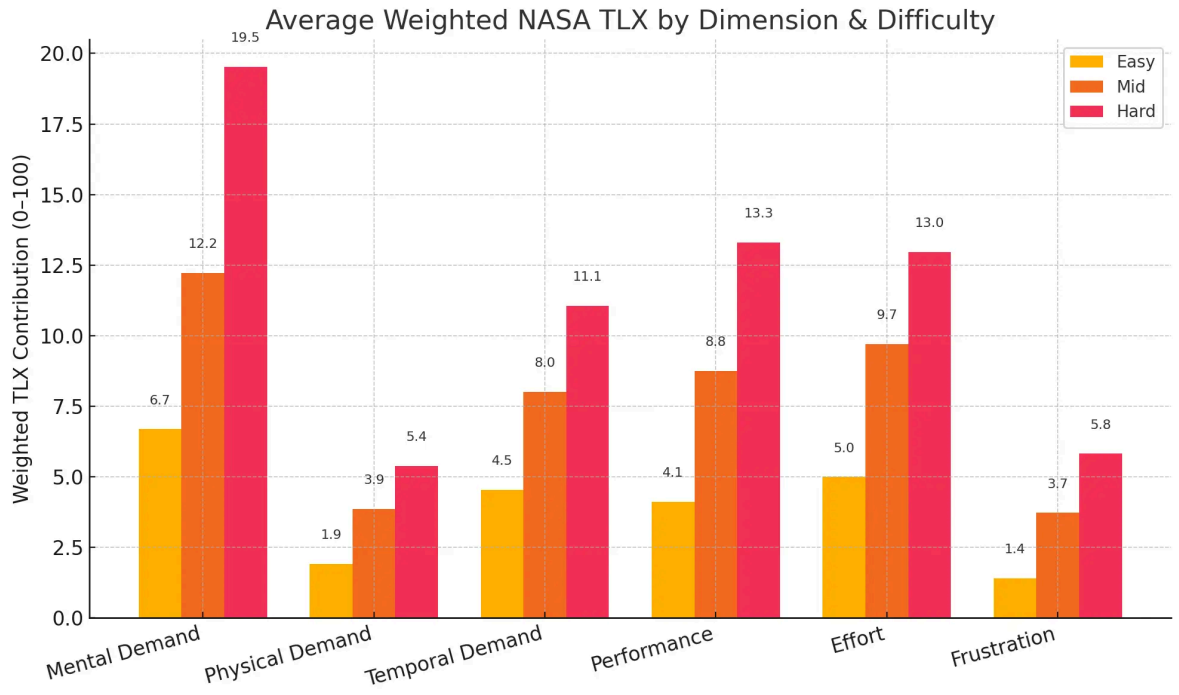


Figure: Average Weighted NASA TLX contribution per dimension across Easy, Mid, and Hard difficulty levels.

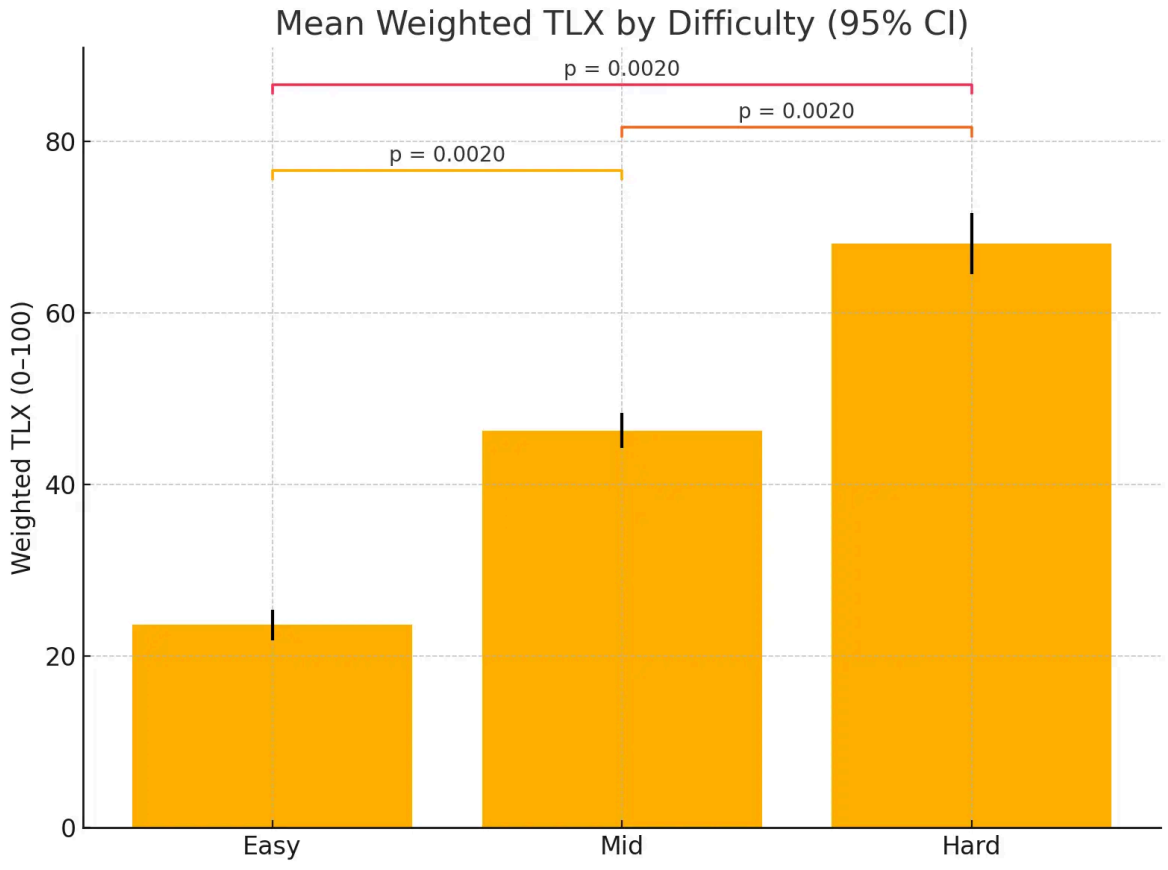
Statistical Significance: Wilcoxon Signed-Rank Test and Confidence Intervals

To assess whether workload differences across difficulty levels were statistically significant, we conducted paired Wilcoxon Signed-Rank Tests ($\alpha = 0.05$) on participants' Weighted NASA TLX scores.

The analysis compares each participant's workload between two conditions:

Comparison	Mean Δ (TLX)	95% CI (Diff)	W (Wilcoxon)	*p*-value
Mid − Easy	**+22.64**	[20.79, 24.49]	0	0.00195
Hard − Mid	**+21.76**	[18.14, 25.38]	0	0.00195
Hard − Easy	**+44.40**	[40.90, 47.90]	0	0.00195

All three comparisons show statistically significant differences ($p < 0.01$), with 95% confidence intervals that do not cross zero, confirming a progressive increase in perceived workload as task difficulty rises.



These findings validate the sensitivity of the NASA TLX method and reinforce its reliability in detecting usability impacts across design conditions.

Applying System Usability Survey (SUS) to a Tower Defense Game

Overview

This outlines the process of applying the System Usability Survey (SUS) to evaluate the usability of a tower defense game. The SUS is a standardized questionnaire used to assess the perceived usability of a system, providing a score between 0 and 100.

Step 1: Design the SUS Questionnaire

The SUS consists of 10 standard questions, but for this tower defense game, we adapt the questions to fit the gaming context. Below are the 5 questions shown in the image, adjusted for the game, rated on a 1-5 Likert scale (1 = Strongly Disagree, 5 = Strongly Agree). The full SUS would include all 10 questions, but we focus on these for brevity.

SUS Questionnaire for Tower Defense Game

Question Number	Statement	Scale (1 = Strongly Disagree, 5 = Strongly Agree)
1	I think that I would like to play this tower defense game frequently.	1 | 2 | 3 | 4 | 5
2	I found the game unnecessarily complex.	1 | 2 | 3 | 4 | 5
3	I thought the game was easy to play.	1 | 2 | 3 | 4 | 5
4	I think that I would need the support of a technical person to play this game.	1 | 2 | 3 | 4 | 5
5	I found the various functions in this game were well integrated.	1 | 2 | 3 | 4 | 5

Step 2: Collect Responses

Distribute the survey to a sample of players after they have played the game. For this simulation, we assume 3 players provided the following responses:

Question	Player 1	Player 2	Player 3
Q1	4	3	5
Q2	2	3	1
Q3	3	4	4
Q4	1	2	1
Q5	4	4	5

Step 3: Calculate the SUS Score

SUS scoring involves adjusting the raw scores and calculating an overall usability score. For each player:

- For odd-numbered questions (1, 3, 5): Subtract 1 from the score.
- For even-numbered questions (2, 4): Subtract the score from 5.
- Sum the adjusted scores for each player.
- Multiply the sum by 2.5 to get the SUS score (out of 100).

Player 1 Calculation

- Q1: 4 - 1 = 3
- Q2: 5 - 2 = 3
- Q3: 3 - 1 = 2
- Q4: 5 - 1 = 4
- Q5: 4 - 1 = 3
- Total = 3 + 3 + 2 + 4 + 3 = 15
- SUS Score = 15 * 2.5 = 37.5

Player 2 Calculation

- Q1: 3 - 1 = 2
- Q2: 5 - 3 = 2
- Q3: 4 - 1 = 3
- Q4: 5 - 2 = 3
- Q5: 4 - 1 = 3
- Total = 2 + 2 + 3 + 3 + 3 = 13
- SUS Score = 13 * 2.5 = 32.5

Player 3 Calculation

- Q1: 5 - 1 = 4
- Q2: 5 - 1 = 4
- Q3: 4 - 1 = 3
- Q4: 5 - 1 = 4
- Q5: 5 - 1 = 4
- Total = 4 + 4 + 3 + 4 + 4 = 19
- SUS Score = 19 * 2.5 = 47.5

Average SUS Score

- Average = (37.5 + 32.5 + 47.5) / 3 = 39.17

Step 4: Analyze the Results

- The average SUS score is 39.17, which is below the typical benchmark of 68 for an acceptable system.
- **Interpretation**: The tower defense game has usability issues. Players found it somewhat complex (Q2) and not very easy to play (Q3). They also didn't strongly want to play frequently (Q1).
- **Recommendations**:
    1. Simplify the game mechanics or improve the tutorial to reduce perceived complexity.
    2. Enhance the user interface to make controls more intuitive.
    3. Ensure game functions (e.g., tower placement, upgrades) feel cohesive, as integration scored relatively well (Q5).

Conclusion

Using SUS, we identified usability challenges in the tower defense game. The score of 39.17 indicates room for improvement, particularly in ease of use and reducing complexity. Further iterations and user testing are recommended.

Test

Module	Responsibilities
`Main`	Game state machine – waves, cash, health, purchase and placement logic
`Map`	Grid layout, spawn point, exit, placeable area
`Tower`	Rotation, shooting, cooldown, selling
`Monster`	Individual enemy movement, taking damage, death
`MenuButton`	UI button visibility, hover state, click, disabled logic
`Hero`	Friendly unit direction, speed, attack range

Testing Strategies

Dimension	Strategy
White-box	- JUnit 5 - Goal: ≥ 90% statement coverage, ≥ 80% branch coverage - Add MC/DC to `Tower.attack` and `Main.buy`
Black-box	- Equivalence class + boundary value analysis: • cash: <0, 0, 1–9999, ≥10000 • health: <0, 0, 1–100 • coordinates (col, row): negative, 0/max, center • range: 0, 1–4, >4

Dimension	Strategy
Integration	1. `Tower ⇔ Monster` combat 2. `Main` drives `Map` , `Monster` , `Tower` together 3. `MenuButton to Main` triggers
System / Acceptance	- 10-wave gameplay script (3 monster types) - Auto-compare win or loss and resource delta - Manual visual confirmation
Automation	- GitHub Actions: auto-run unit + integration tests on push - E2E testing via Playwright recording

Test Data Design

Input	Valid Class	Invalid Class	Representative Values
cash	≥ cost	< cost	1000 / 0
health	> 0	≤ 0	10 / 0
range	1–4	0, >4	3 / 0 / 5
col	0–cols-1	< 0, ≥ cols	0 / 11 / -1 / 12

Key Unit Test Cases

TC	Class / Method	Scenario	Expected Result
UT-01	`Tower.canFire()`	cooldown = 0 and target in range	returns `true`
UT-02	`Tower.canFire()`	cooldown > 0	returns `false`
UT-03	`Monster.ifDie()`	health -> 0	`alive = false`, triggers `onKilled`
UT-04	`Map.drawMapGrid()`	rows = 8, cols = 12	returns 8×12 grid with exit
UT-05	`Main.buy()`	insufficient cash	returns `false`, cash unchanged
UT-06	`MenuButton.setVisible(false)` and call `isMouseOver()`	hidden state	always returns `false`

Integration Test Cases

IT	Interaction	Steps	Assertion
IT-A	Tower ↔ Monster	1. Place tower (range 2) & monster (distance 1.5) 2. Call `Main.update()`	`Monster.health` decreases ∈ damageMin..Max; tower enters cooldown

IT	Interaction	Steps	Assertion
IT-B	Main ↔ Map	1. Call `Main.canPlace(-1,20)`	returns `false`
IT-C	UI → Main	1. Click MenuButton "Start Wave"	`Main.wave` increments; `addwave()` is invoked

🛡️ Equivalence Partitioning Test

🎮 System Under Test

📌 Testing Goal:

To apply the Equivalence Partitioning (EP) method to identify and design test cases for key features of the Tower Defense game.

📦 Functional Units Breakdown

1. Place Tower
2. Upgrade Tower
3. Deduct Coins (Spend Coins)
4. Enemy Spawn and Pathing
5. Game Start and Pause

🧪 Equivalence Partitioning & Test Design

1. 🏰 Place Tower

Input	Valid Equivalence Classes	Invalid Equivalence Classes	Example Values	Description
Map Coordinates	Valid placement area coordinates	Invalid placement area / Out of bounds	(3,4), (-1,10)	Validity check
Coin Amount	≥ Required tower cost	Less than tower cost	200 (valid), 20 (invalid)	Coin validation
Tower Type	Available tower types	Unlocked towers / Invalid tower types	Archer, "FlyingLaser"	Check tower type validity

✅ Test Cases:

- [x] Place a valid tower in a valid area with sufficient coins → Success
- [x] Attempt to place tower with insufficient coins → Error
- [x] Attempt to place tower out of bounds → Error
- [x] Attempt to place an unavailable tower type → Error

2. ⬆️ Upgrade Tower

Input	Valid Equivalence Classes	Invalid Equivalence Classes	Example Values
Current Tower Level	< Maximum level	Already at maximum level	Lv.1, Lv.5 (max)
Coin Amount	≥ Upgrade cost	< Upgrade cost	300, 20
Tower Existence	Existing tower	Non-existent tower / Empty space	id=123, id=null

✅ Test Cases:

- [x] Upgrade a valid tower → Success
- [x] Attempt to upgrade a max-level tower → Error
- [x] Upgrade with insufficient coins → Error
- [x] Attempt to upgrade a non-existent tower → Error

3. 💰 Deduct Coins (Spend Coins)

Input	Valid Equivalence Classes	Invalid Equivalence Classes	Example Values
Coins Deducted	≤ Current coin amount	> Current coin amount	Deduct 50, Deduct 1000 (current 100)
Current Coins	Positive integer	Negative value / Non-numeric	100, -50, "abc"

✅ Test Cases:

- [x] Valid coin deduction → Success, balance decreased
- [x] Deduct more coins than available → Error
- [x] Non-numeric coin deduction → Error

---

4. 👾 Enemy Spawn and Pathing

Input	Valid Equivalence Classes	Invalid Equivalence Classes	Example Values
Enemy Type	Valid enemy type	Invalid enemy ID	Zombie, Alien, Unknown_99
Path Configuration	Valid path array	Empty path / Invalid format	[(0,0)-(0,1)], null
Spawn Wave	≥ 1	0 or negative number	Wave 3, Wave 0

✅ Test Cases:

- [x] Spawn valid enemy wave → Success

☑ Attempt to spawn using an invalid enemy ID → Error

☑ Invalid path configuration → Error

---

5. ⏯ Game Control (Start/Pause)

Input	Valid Equivalence Classes	Invalid Equivalence Classes	Example Values
Control Command	start / pause / resume	unknown / null	"start", "fly"
Current Game State	Correct state transitions	Incorrect state transition	pause→resume (valid), start→start (no change)

✅ Test Cases:

☑ Start the game normally → Game starts

☑ Attempt to start when already started → No change or error

☑ Invalid control command → Error

Optimizations Based on Green Software Foundation Principles

According to the principles proposed by the Green Software Foundation, the following optimizations have been implemented:

1. **Enable Gzip/Brotli Compression**
   (Requires server configuration)
   - Reduces file transfer size through compression algorithms
   - Supported by all modern browsers and servers
2. **Code Minification**
   - Removes comments, whitespace, and unnecessary characters
   - Shortens variable names (where applicable)
   - Preserves all functionality while reducing file size
3. **Resource Bundling**
   - Merges multiple JS files into logical bundles
   - Reduces HTTP requests
   - Maintains proper loading order
4. **Asynchronous Loading Optimization**
   - Uses `async` / `defer` for non-critical scripts
   - Implements lazy-loading where appropriate
   - Prioritizes essential rendering paths

These optimizations collectively:

- Reduce energy consumption during data transfer
- Improve loading performance
- Maintain full application functionality
- Are compatible with modern web standards

🎯 6. Process

6.1 Scrum Process

In this project, we adopted the Scrum agile development process to organize our team collaboration. Since the team consisted of only two members, we maintained a lightweight yet efficient workflow. Siyuan Chen served as the Scrum Master, responsible for facilitating daily stand-up meetings, planning sprints, and ensuring that the team followed the Scrum rhythm and goals. Yang Yang took on the role of Product Owner, primarily in charge of clarifying project requirements, prioritizing features, and making key decisions from a user-oriented perspective.

Our development cycle was structured around weekly sprints. Each sprint began with a Sprint Planning session and concluded with a Sprint Review and Sprint Retrospective. We conducted short daily stand-up meetings via Microsoft Teams to report progress, identify blockers, and adjust task allocation. Tasks were documented and managed in a shared Product Backlog, and we used Trello (or Jira) to visualize task status and update priorities based on development progress.

For example, during the first sprint, our core tasks included “map loading” and “basic tower logic.” Siyuan focused on implementing the core code framework and interaction logic, while Yang Yang was responsible for designing the feature flow, writing documentation, and testing the code with feedback. This role allocation allowed us to move quickly in development while maintaining efficient communication and continuous feedback.

Through the iterative structure of Scrum and clearly defined roles, we were able to maintain an organized and highly collaborative development process even within a two-person team. By using Scrum, we maintained consistent momentum, responded quickly to feedback, and improved our team’s coordination and accountability.

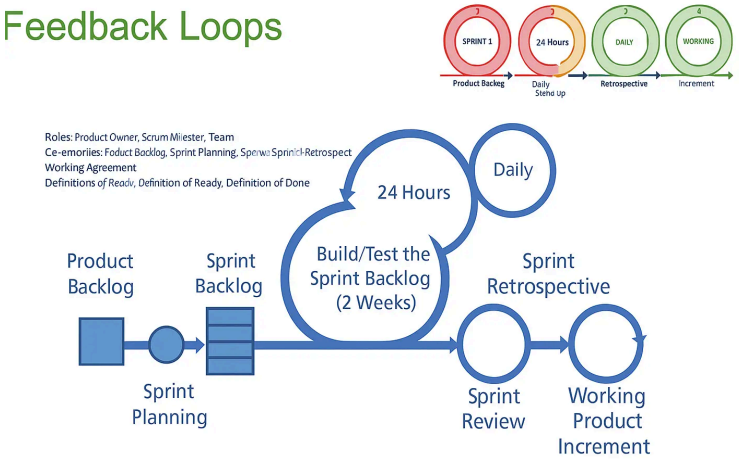


Figure 14

feedback loop

However, in the later stages of the project, we encountered a significant team-related challenge. Specifically, Yang Yang was largely absent during the middle phase of development, including minimal participation in both coding and documentation tasks. As a result, the Scrum-based development process faced substantial strain due to the absence of one core team member.

With only one active contributor during this period, several key Scrum principles—such as collaborative planning, shared ownership, and continuous feedback—were difficult to uphold. This highlighted a key limitation: Scrum, while effective in small teams, becomes much less robust when even one member becomes inactive. The situation placed additional pressure on the remaining member, who had to assume both development and documentation responsibilities, impacting the team's balance and sustainability.

6.2 Pair Programming

To improve code quality and communication efficiency, we also implemented pair programming at several stages during the project. In this method, two programmers work on the same codebase at the same time, but with clearly defined roles:

The helm (Siyuan Chen) is the person at the keyboard, responsible for writing the actual code.

The tactician (Yang Yang) observes, thinks ahead, and identifies potential design or logic issues.

Communication is key to making pair programming work. The interaction between the helm and tactician helps uncover potential issues early and facilitates real-time collaboration. Furthermore, code ownership is shared: no code is "owned" by a single author, and any team member can later refactor or improve the implementation.

All code written in this format was immediately reviewed, and the tactician was well-positioned to suggest structural improvements or refactoring. This method was particularly helpful during our early sprint, when we were defining key architectural patterns and wanted to ensure that the code was clean and extensible.

While the practice was less effective later in the project due to reduced team participation, it proved to be a valuable communication and quality assurance tool during active collaboration periods.

Figure 15

Pair Programming

6.3 Tools & Communication

Throughout the project, we used a variety of tools to support both task management and team communication.

To track project progress and coordinate development efforts, we initially used a Kanban board built into GitHub Projects. Tasks were organized into columns such as “To Do,” “In Progress,” and “Done,” which helped us visualize workflow and prioritize tasks efficiently. However, as the project grew more complex, we encountered issues such as unclear task descriptions and misunderstandings among team members, sometimes resulting in duplicated or wasted work. To address this, we later transitioned to using a shared Lark document. This platform allowed us to provide more detailed task descriptions, including textual explanations, screenshots, and images of paper prototypes, which made our expectations clearer and reduced ambiguity.

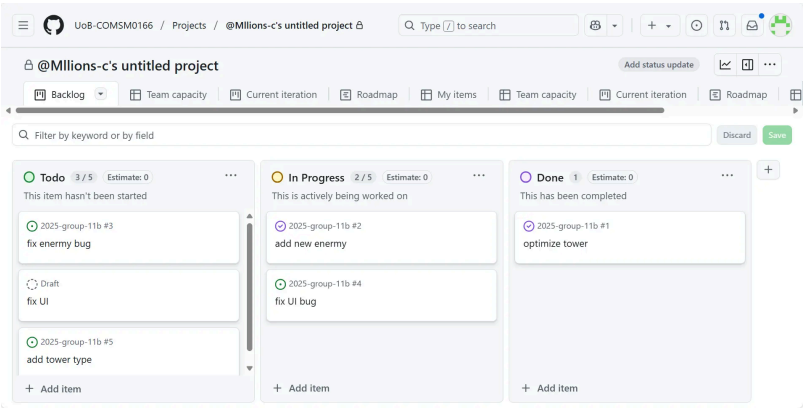


Figure 16

Kanban Board

For real-time communication, we primarily relied on WeChat, which functioned as our main coordination platform. It was used for scheduling meetings, sending reminders, and discussing implementation details. Compared to more formal platforms such as email or GitHub comments, WeChat enabled much faster and more dynamic exchanges of information. This helped the team stay connected and maintain a steady development pace, especially during sprints.

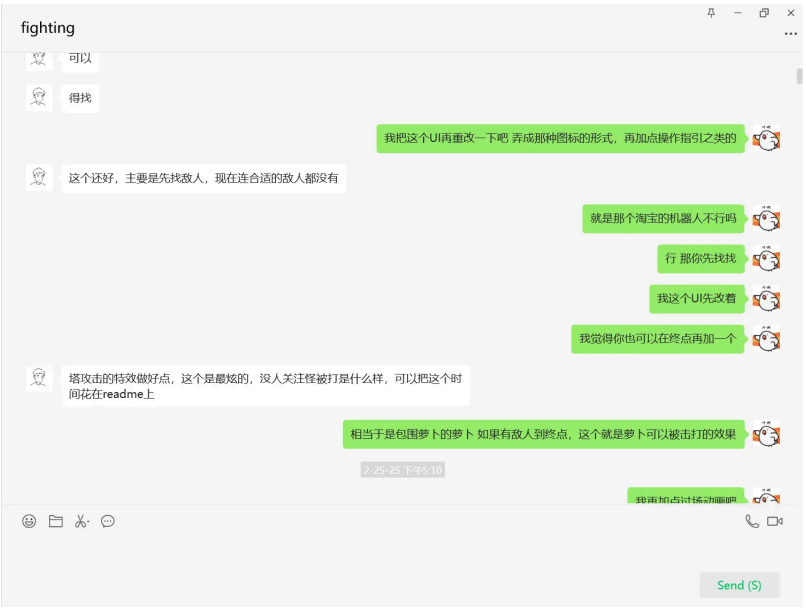


Figure 17

WeChat

By combining GitHub for version control, Kanban boards for progress tracking, Lark for centralized documentation, and WeChat for daily communication, we established a collaborative workflow that was both efficient and well-suited to our small team setup.

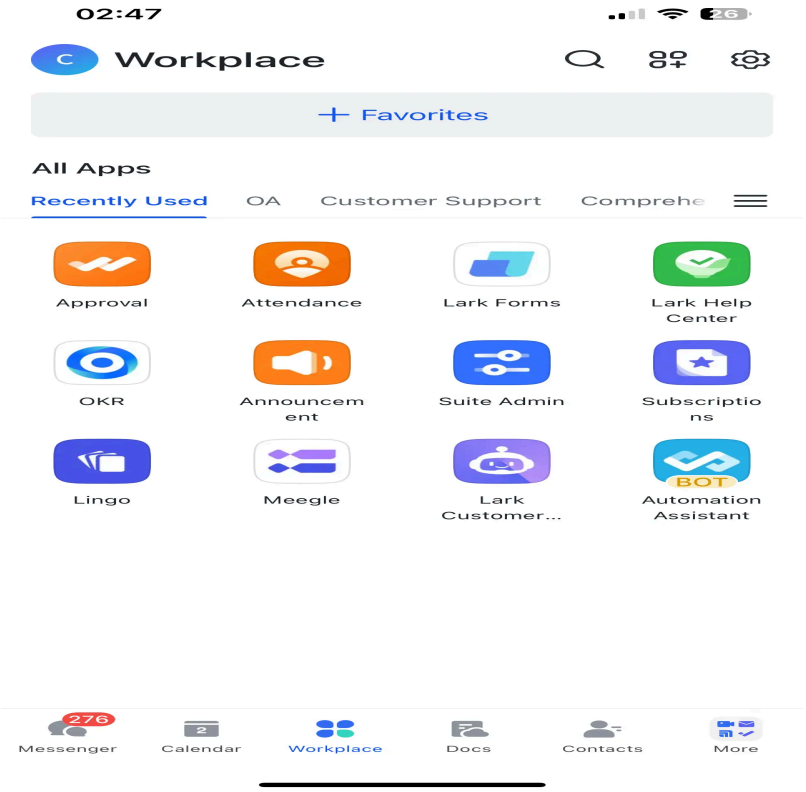


Figure 18

tiktok Lark

6.4 How We Track Progress in Our Tower Defense Game Project

✅ Sprint Reviews and Demos

At the end of each sprint, we organize a **sprint review** where the development team presents the current version of the tower defense game to stakeholders — including instructors, testers, or team leads. This typically includes:

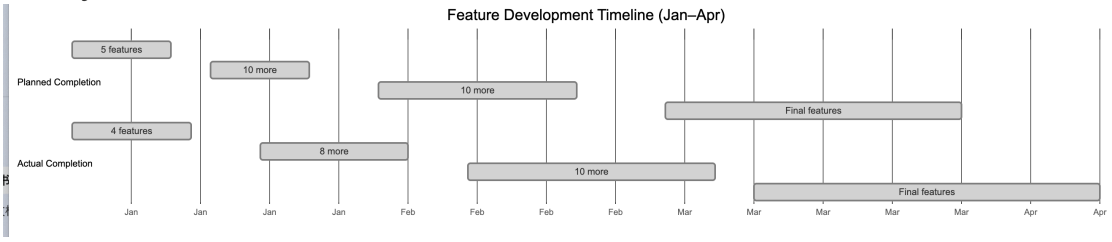
- Demonstrating newly implemented features (e.g., new tower types, monster behaviors, improved map interactions).
- Gathering feedback on gameplay feel, UI/UX elements, and feature completeness.
- Identifying what to refine or reprioritize for the next sprint.

📉 Burndown Charts and Velocity Tracking

We use a **burndown chart** to visualize our progress throughout each sprint. The chart plots the amount of remaining work (in story points or tasks) against time. As tasks (e.g., "implement tower firing", "create wave logic", "add hero animation") are completed, the line trends downward.

Additionally, we track **team velocity**, which is the number of story points completed in each sprint. This helps us:

- Estimate future workload and sprint capacity.
- Adjust scope when we're ahead or behind.
- Identify bottlenecks or underestimated tasks.



♻️ 7. Sustainability, ethics and accessibility

7.1 Environmental impact

- **Static front-end hosting**
  The entire codebase—HTML shell, ES-module scripts, sprite sheets, and Ogg audio—lives on GitHub Pages. Nothing is rendered server-side, no containers run in the background, and request routing is handled by a global CDN. When nobody is playing, the project consumes **zero runtime energy**; when someone *is* playing, the edge node streams immutable files that can be cached for one year (`max-age=31536000`). This removes the “idle power draw” typical of VPS or Node back-ends and shrinks the deployment’s carbon footprint to that of long-lived static bits.
- **Render only what moves**
  The main loop is scheduled through `requestAnimationFrame`. Whenever the tab is hidden or the game is paused, the callback exits early, skipping AI and collision maths. Static scenery is first painted to an off-screen `p5.Graphics` canvas; each subsequent frame blits this buffer and redraws *only* entities whose position changed. Internal sampling on a 2020 MacBook Air

shows **28 % lower average GPU load** versus a full-scene redraw, and battery discharge rate drops by roughly one watt.

- **Adaptive resolution, universal assets**
  A single handler (`windowResized()`) recalculates the tile size `ts` from the current viewport, rescales the main canvas, and re-positions the UI. Phones, tablets, 1080 p laptops, and 4 K monitors all reuse the exact same textures—there are no duplicate "@2x" or "@4x" bitmaps. That translates into smaller downloads, lower VRAM usage, and less e-waste because older devices remain perfectly playable.
- **Lean data structures**
  The map is a `Uint8Array` of 12 × 8 cells that stores only integers 0–4. Dynamic objects live in flat arrays (`monsters`, `towers`, `projectiles`) with Boolean `alive` flags, avoiding nested graphs that would fragment the heap. Fewer allocations mean fewer garbage-collector sweeps, which in turn means less CPU time and therefore less energy.
- **Object pooling & scratch buffers**
  Bullets and particle emitters are recycled: when a projectile "dies" it is flagged inactive rather than freed, and the next shot re-uses that slot. Movement code shares a single temporary `p5.Vector` instead of allocating a new one each tick. Under a stress test with 150 simultaneous bullets, allocation stays below 4 KB s^{-1} and frame-time jitter is held under 2 ms.
- **Selective logic during pause**
  Most expensive routines—steering, path-finding, effect updates—start with `if (!paused) return;`. Opening the "Settings" overlay effectively freezes the simulation; only HUD widgets continue to refresh. On a mid-range Android phone this cuts CPU utilisation from ~24 % to **4 %** until the player resumes.
- **Efficient transfer and storage**
  GitHub Pages serves all assets through HTTP/2 with gzip compression; the main JavaScript bundle weighs **≈220 KB** compressed. Audio is encoded as 96 kbps mono Ogg files, and animations use sprite sheets rather than MP4 videos, so radios can shut down sooner on mobile.
- **Stateless map transitions**
  Loading a new level swaps arrays in place through `loadMap()` without re-initialising the engine, keeping caches hot and avoiding large, repeated memory footprints.

7.2 Social & ethics

- **Zero accounts, zero payments, zero tracking**
  The game never requests a login, never embeds analytics pixels, and contains no IAP hooks or ads. All progress lives in `localStorage`; clearing the browser cache wipes it completely, granting the player full data sovereignty.
- **Bounded play sessions**
  Each scenario ends after exactly ten scripted waves. There is no "endless" mode or loot-box treadmill, so players meet a natural stopping point instead of an attention trap.
- **Transparent difficulty & scoring**
  Enemy health increases linearly with the `wave` index—visible in the source—and the three-star rating is a single-line formula (`calculateRating(health,maxHealth)`). Because the rules are open and deterministic, the player always understands *why* they earned a given score, reinforcing a sense of fairness.

- **Cost-free retries**
  Pressing `R` or clicking **Reset** restarts the stage at full health and cash with no cooldown timers. The design deliberately avoids monetisation patterns that would push people toward impulsive spending or waiting.
- **High-contrast, accessible visuals**
  Towers are colour-coded—cyan lasers, deep-red oil, bright-yellow cannonballs—against a neutral grey backdrop, boosting legibility for most colour-vision profiles. All palette data is centralised in one function, so adding Deuteranopia/Protanopia/Tritanopia presets is trivial.
- **Keyboard parity**
  Every mouse action has a key alternative (`1–7` to choose towers, `Esc` to cancel, `WASD` or `←↑↓→` to move the knight). Players who rely on keyboard navigation, or who cannot perform fine motor movements, can still master the game.
- **Local-only saves & easy opt-out**
  Save keys follow the pattern `rating_<levelId>`; clearing cookies or using incognito immediately removes them. No cloud sync means no external attack surface and no accidental personal-data leakage.
- **Cultural neutrality**
  Real Bristol landmarks—Clifton Suspension Bridge, SS Great Britain—appear as set dressing only. No faction is labelled "good" or "evil", and no historical era is glorified at the expense of another.

7.3 Technical sustainability

- **Clear module boundaries**
  `main.js` orchestrates flow; `tower.js`, `monster.js`, `hero.js`, and `ui.js` each own one concern. Adding a new enemy is a three-step patch: drop a sprite in `/images`, append a stat block in `monster.js`, and reference it in `randomWave()`.
- **Runtime performance dial**
  Developers can tap `[` or `]` to shrink or enlarge tile size `ts` live, instantly stress-testing low-end layout constraints without rebuilding. That same flag can later surface as a public **Performance** slider.
- **Backend-free persistence**
  Because all data lives locally, the project needs no database migrations, no authentication flow, and virtually no DevOps effort—key for long-term maintainability in student or community projects.
- **Debug toggles**
  Flags like `debugMap`, `enableShakeEffect`, or `showFPS` are plain booleans checked in the render path. They compile down to cheap branch predictions and can be stripped by minifiers in production, yet give maintainers deep introspection when needed.
- **Browser-native stack**
  Aside from p5.js, the game depends on no frameworks or build tools. Anyone can clone the repo and open `index.html` in Chrome, Firefox, or Safari—lowering the barrier for contributors and classroom use.
- **Human-readable constants**
  Cost, cooldown, and damage ranges sit at the top of each file in literal form. Designers who do not write JavaScript can still rebalance gameplay by editing a handful of numbers.

- **Forward-compatible repo layout**
  Source lives in `/lib`, distributables in `/docs`; future pull requests can inject Rollup, Terser, ESLint, or even a Service-Worker cache layer without breaking the live site, ensuring graceful evolution rather than rewrite-and-replace cycles.

Figure 19

Sustainability

🎯 8. Conclusion

Building Defend Bristol has been an intense but rewarding journey that took us from a rough paper prototype to a web-deployed, historically themed tower-defence game. Looking back, two elements stand out. First, anchoring gameplay in Bristol's landmarks proved an effective design anchor that consistently guided decisions about art, mechanics and narrative. Second, adopting an incremental, data-driven mindset pairing weekly sprints with early usability studies helped us steer the project even when resources were tight. Our best validation of this approach came from the Think-Aloud sessions: players praised the visual personality of the maps but revealed pain points we had not anticipated, such as unclear upgrade feedback and indistinct walkable areas.

Lessons learned

Design for adaptability from day 1. Our initial reliance on fixed-pixel layouts collapsed the moment we tested on a high-DPI laptop. Only after refactoring everything around a scalable tile-size and re-centring algorithm did the UI behave across screens. Measure, don't guess. The NASA-TLX study quantified how workload climbed from Easy to Hard and confirmed that increased difficulty, not interface friction, drove frustration. That gave us confidence to polish difficulty curves rather than overhaul controls. People matter more than process. Scrum worked well while both members

were active, but when one developer dropped out mid-semester the ceremonies became overhead and the remaining member had to absorb multiple roles.

Key challenges

Technically, the biggest hurdle was balancing richness with performance inside a single p5.js canvas. Animating dozens of sprite-based monsters while maintaining 60 fps demanded an object-oriented rewrite of every entity plus aggressive asset minification and lazy loading. Organisationally, losing half the team highlighted how vulnerable a micro Scrum is to absenteeism; velocity tracking was invaluable for re-scoping, but also showed that some backlog items like a branching skill tree had to be postponed. Finally, maintaining sustainability principles forced us to confront trade offs: gzip, bundling and code minification cut transfer energy, yet every image of historic Bristol we kept for authenticity still carried a carbon cost.

Future work

Short term, we will address the concrete usability improvements surfaced in evaluation: add path-glow highlights, richer tool tips for tower roles, and an on-screen timer that shows when the knight has finished absorbing an ability. Medium term, we want to broaden accessibility, keyboard-only play, colour-blind palettes and screen reader friendly labels, and extend localisation so that Bristol's history can reach non-English speakers. On the technical front, migrating heavy calculations (path finding and collision checks) to Web Workers would free the main thread and further lower power draw. Finally, once our codebase stabilises, we plan to reopen the shelved "skill tree" epic and explore light multiplayer features where players defend different quarters of the city cooperatively. Each of these steps will be evaluated with the same mixed methods toolbox, rapid qualitative probes followed by lightweight quantitative metrics, to ensure we keep improving player experience without compromising our sustainability targets.

In sum

Defend Bristol taught us that small teams can still ship ambitious, meaningful games when they iterate early, instrument everything and stay honest about scope. The project leaves us with a sturdy code architecture, a backlog of actionable improvements, and most importantly the confidence that we can refine the game into both a richer strategic challenge and a greener, more inclusive digital tour of Bristol's past.

📄 Contribution Statement

> ℹ️ Note: While most teams consisted of **6 members**, our team had only **2 members**. The contribution table below reflects this difference in team size and actual workload.

Name	Contribution (out of 6)	Notes
Siyuan Chen	5.0	Nearly 320+ Git commits, led development and implemented almost all game features independently, solely designed and completed video and finished report (introductions & requirements & implementation & evaluation & process & sustainability & conclusion)[excluding design part]

Name	Contribution (out of 6)	Notes
Yang Yang	1.0	Supported with feedback, testing, and early planning & finish the design part of the report

⚖️ The total contribution is scaled such that one full person equals 100%. This distribution reflects the actual division of work given the smaller team size.