DEPARTMENT OF INFORMATION TECHNOLOGY
CHAITANYA BHARATHI INSTITUTEOF TECHNOLOGY (A)
2023-2024

GRID CLASH! A 2-D GAME

- **Title:** Grid Cash
- **Name of students:** K. Sai Shiva Varma (160124737185)
  T. Yaswanth (160124737200)
  T. Jugal Kishore Reddy(160124737202)
- **Institution and Department:** IT Department, CBIT
- **Guide/Supervisor's Name:** Ms. P. Kiranmaie (Assistant Professor, IT)
- **Submission Date:** 19-05-2025

TABLE OF CONTENTS

1. ABSTRACT

The Grid Clash project is a 2D arena battle game developed using C++ and various data structures to create a dynamic, fast-paced combat experience. The game focuses on precision movement, strategic dodging, and tactical shooting within a confined grid. The project aims to demonstrate efficient use of data structures such as queues, vectors, and entities, while also integrating complex attack patterns and AI behavior. The game is divided into two challenging levels: a training mode (Level 1) for practicing movement and shooting, and an intense boss fight (Level 2) featuring advanced attack mechanics. The player can move freely within the grid, shoot projectiles, and perform rapid dashes to evade incoming attacks. Meanwhile, the boss can unleash powerful laser strikes, radial shockwaves, and chasing projectiles, requiring quick reflexes and strategic positioning from the player.

2. INTRODUCTION

This project is a 2D console-based game developed in C++ that combines real-time gameplay with core data structure concepts. The game is divided into two levels: a Flappy Bird-style training level and a final boss fight level. It uses a grid-based system to represent the game map, allowing precise control over entity movement and interactions. Key features include dynamic player and enemy behavior, bullet mechanics, health and stamina systems, and various attack patterns, all of which are managed using arrays, queues, and custom logic. The primary goal of this project was to create an engaging game while applying fundamental programming and data structure skills.

1. Game Overview:

Grid Clash is a 2D grid-based combat game developed using C++ and data structures like queues and vectors. The game focuses on strategic movement, precise shooting, and intense boss battles, offering a challenging gameplay experience. The player must navigate through tight spaces, avoid powerful attacks, and carefully manage stamina to survive and defeat the boss.

The boss AI uses random movement and line-targeted laser attacks, demanding quick reflexes and tactical positioning from the player. Bullets are managed through queues to simulate real-time projectile movement, while the grid system ensures structured positioning and collision detection. Player and boss health are visually represented, adding urgency to every encounter. The game features different attack patterns, with proximity-based damage, promoting both aggressive and defensive playstyles. With each level, the difficulty ramps up, testing the player's mastery over grid control and timing.

2. Core Features:

a) Real-time player and boss movement with dynamic health and stamina systems.

b) Advanced projectile mechanics using FIFO queues for efficient shot handling.

c) Boss abilities including laser strikes, shockwaves, and targeted projectiles.

d) Dash mechanics for rapid player repositioning and strategic evasion.

e) Two unique levels, each with distinct challenges and escalating difficulty.

2.3 Objective:

The primary objective of Grid Clash is to provide a fast-paced, skill-based gameplay experience that emphasizes precise movement, quick decision-making, and resource management. The project aims to showcase efficient use of data structures to handle complex game mechanics and create a responsive combat system.

2.4 Scope:

The following defines the scope for this 2D grid clash game project:

a) Core Functionality:

- **Level1 (Training Mode):** Focuses on player movement and shootings.
- **Level 2 (Boss Fight):**Introduces advanced boss AI advanced combat techniques.
- **Future Expansion**: Potential additions include multiplayer support, power-ups, and enhanced enemy AI for a more challenging experience.

b) Advanced Features:

- **Laser Strike**: The boss can unleash powerful laser attacks that cover entire rows and columns, requiring the player to quickly reposition to avoid damage.
- **Shockwave Attack:** Expanding and contracting radial shockwaves that deal area-of-effect damage, creating unpredictable movement patterns.
- **Chasing Projectiles**: The boss fires targeted missiles to track the player.
- **Random Movement**: The boss can switch between aggressive pursuit and random wandering, making its movement patterns unpredictable.

3. METHODOLOGY

The methodology used in the Grid Clash game involves designing the game environment using a 2D array to simulate a grid-based map. Object-oriented programming principles are applied by creating classes for the player, boss, bullets, and game logic, promoting modularity and code reuse. Movement and interactions are handled using loops and conditionals that check and update the grid state. Data structures like queues and arrays help manage enemy behavior and bullet movement efficiently. The game loop continuously updates player input, enemy actions, and collision checks. File handling is used for storing high scores or level data. This approach ensures a structured and scalable codebase.

3.1 Data Structures Used:

a) QUEUE's: Used to manage player and boss projectiles efficiently using the FIFO (First- In- First-Out) principle. Allows easy addition and removal of projectiles, ensuring fast real-time gameplay. Manages player shots (player shots) and boss shots(boss shots) separately to prevent overlap and collision issues. Used for projectile movement and collision detection to handle continuous shooting mechanics.

b) VECTOR's: Used to track laser warning tiles (laser tiles) for the boss's powerful cross laser attack. Manages expanding and contracting shockwave patterns for area-of-effect attacks. Efficiently handles dynamic wave growth and contraction, allowing real- time updates. Allows quick access and modification of active wave data without complex memory management. Used to store Wave objects, providing flexibility in managing multiple wave patterns simultaneously.

c) ARRAY's: Used to represent the game grid where each cell holds information about players, enemies, bullets, and obstacles. This allows for efficient tracking and updating of positions during movement or combat. Arrays help manage the layout of the game world and make it easier to implement features like collision detection and bullet trajectories. They also enable quick access to elements for rendering and game logic decisions.

c) ENTITIES (Player and Boss): Player and Boss are implemented as objects with attributes like position, health, and stamina. Simplifies state management by encapsulating data and functions for each entity. Facilitates modular design, making it easier to add new features like health regeneration and movement restrictions. Supports efficient collision detection and interaction handling between player and boss.

3.2 Header Files Used:

a) <iostream>: Enables input and output operations, allowing interaction with the console through standard streams like cout and cin.

b) **<conio,h>:** Provides functions such as _kbhit() and _getch() which allow capturing keyboard inputs instantly without requiring enter key. This is crucial for real-time game controls.

c) **<windows.h>:** Utilized for system-specific operations like controlling the cursor position and, clearing the screen, and creating time delays using functions like Sleep().

d) **<cstdlib>** and **<ctime>:** These headers are used for generating random numbers, important for adding unpredictability to enemy movements and game events. ctime seeds the random number generator with time.

e) **<vector>, <queue>** and **<stack>:** These standard template library (STL)data structures facilitate dynamic memory management and efficient implementation of algorithms such as BFS for enemy AI, pathfinding, and game state management

f) **<algorithm>:** Provides utility functions like sorting and searching, which support various game functionalities where data manipulation is required.

3.3 Game Mechanisms and Working:

a) Player Movement:

- The player moves using keyboard inputs captured in real-time via functions from <conio.h>.
- Movement is restricted within the game grid to prevent out-of-bound errors.

a) Enemy Behaviour:

- Enemies move randomly or follow simple AI algorithms (e.g.,BFS) using data structures like queues or stacks.
- Random movements and actions are generated using functions from <cstdlib>and <ctime>to seed randomness.

c) Game Display and Animation:

- The console screen is updated continuously using cursor control functions from <windows.h>.
- The Sleep()function is used to control the speed of animations and gameplay.

c) Collision Detection:

- The game checks for collisions between player, enemies, and bullets by comparing their positions in the grid.
- When a collision is detected appropriate actions such as health reduction or game over are triggered.

e) Health and Scoring System:

- Player and enemy health are tracked with variables and updated based on hits or attacks.
- Scores increase when theplayer successfully defeats enemies or completes objectives.

f) Game Loop:

- The game runs inside a continuous loop, checking for inputs, updating game states, moving entities, and rendering the screeneach frame.
- The loop exits when the player wins, loses, or manually quits.

4. IMPLEMENTATION

The Grid Clash 2D game was implemented using C++ in a Windows environment, with development carried out in an IDE like Code::Blocks for easier debugging and compilation. The game logic is structured using classes to represent players, enemies, and bullets, ensuring a clean and modular design. A 2D array forms the game grid, enabling efficient position tracking and updates. Real-time gameplay is achieved through direct keyboard input and smooth screen rendering. Randomization adds unpredictability to enemy behavior, while algorithms like BFS enhance enemy AI. Overall, the project combines logical structuring, real-time interaction, and data handling for a dynamic gaming experience.

1. Programming Environment

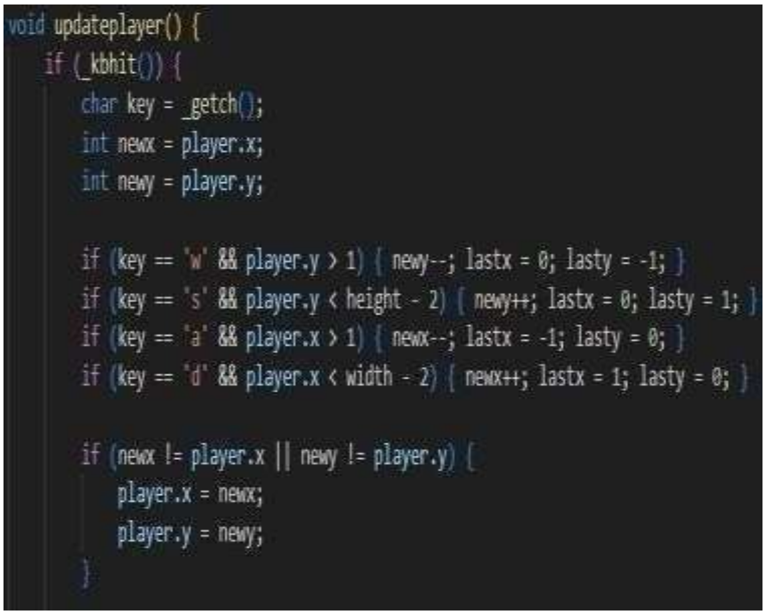
This 2D game project is developed using **C++**, typically writtenin an **IDE** like **Code::Blocks**,**Dev-C++**,or **VisualStudioCode**. It uses **STL headers** like <vector>, <queue>, and <stack>for datamanagement, along with **<conio.h>**forreal-time input and **<windows.h>** for screen control. Compilation is done with **G++** or **Clang**, and the game runs in a console window, providing a straightforward environment for 2D grid-based gameplay.

2. Code Overview:

The code implements a 2D console game in **C++**, using data structures like **vectors**, **queues**, and **stacks** for efficient game logic. It handles player and enemy movements, collision detection, and game state management through a continuous game loop. The code also uses **real-time input** and **screen control** for smooth gameplay.

a) Player Movements:

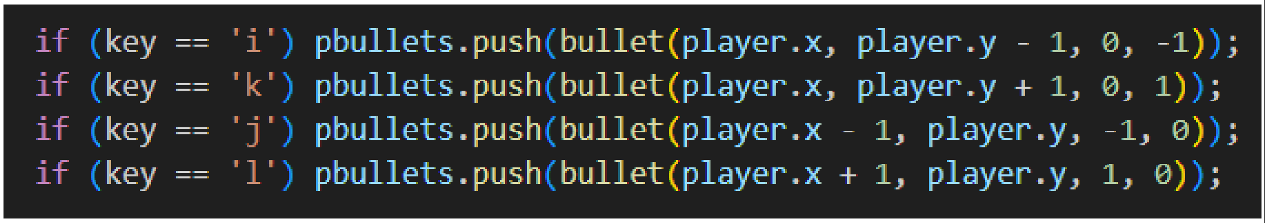
Player movement is handled within the **updateplayer()** function using the **WASD** keys for directional movement. The player's **x** and **y** coordinates are updated based on the key pressed, with boundary checks to prevent moving outside the grid. The **lastx** and **lasty** variables store the most recent movement direction, allowing for directional dashing. Dashing is triggered with the **spacebar**, moving the player multiple cells in the last direction if enough **stamina** is available. Stamina regenerates gradually, and the player can also shoot bullets in the **up**, **down**, **left**, and **right** directions using **IJKL** keys.



```
void updateplayer() {
    if (_kbhit()) {
        char key = _getch();
        int newx = player.x;
        int newy = player.y;

        if (key == 'w' && player.y > 1) { newy--; lastx = 0; lasty = -1; }
        if (key == 's' && player.y < height - 2) { newy++; lastx = 0; lasty = 1; }
        if (key == 'a' && player.x > 1) { newx--; lastx = -1; lasty = 0; }
        if (key == 'd' && player.x < width - 2) { newx++; lastx = 1; lasty = 0; }

        if (newx != player.x || newy != player.y) {
            player.x = newx;
            player.y = newy;
        }
```

b) Bullet Mechanism:

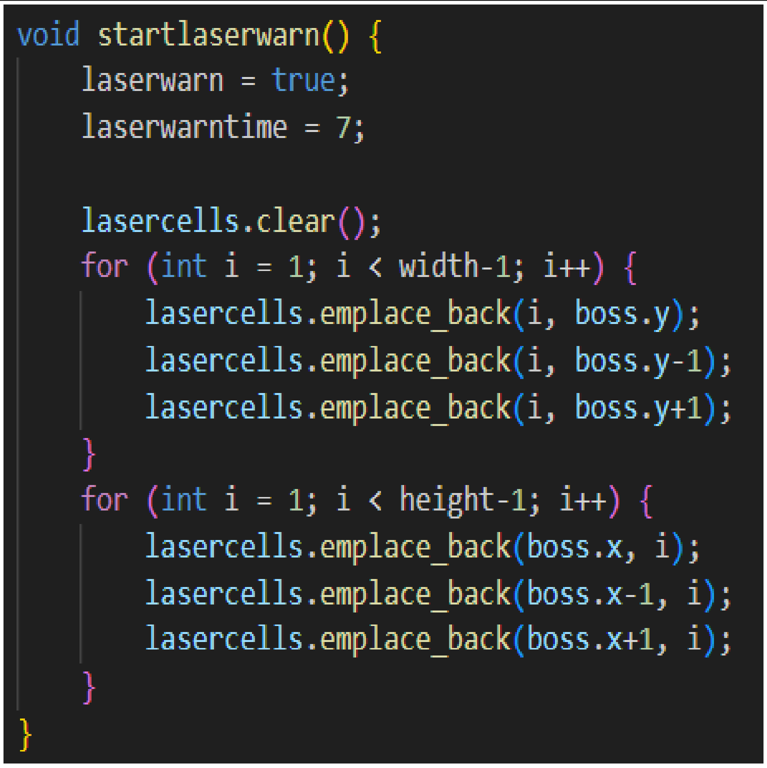
The bullet firing mechanism uses the **IJKL** keys to shoot in four directions. When a key is pressed, a **bullet** object is created with the player's current position and a specific direction, like **I** for up, **K** for down, **J** for left, and **L** for right. These bullets are then added to the **pbullets** queue for processing.



```
if (key == 'i') pbullets.push(bullet(player.x, player.y - 1, 0, -1));
if (key == 'k') pbullets.push(bullet(player.x, player.y + 1, 0, 1));
if (key == 'j') pbullets.push(bullet(player.x - 1, player.y, -1, 0));
if (key == 'l') pbullets.push(bullet(player.x + 1, player.y, 1, 0));
```

c) Laser Warning:

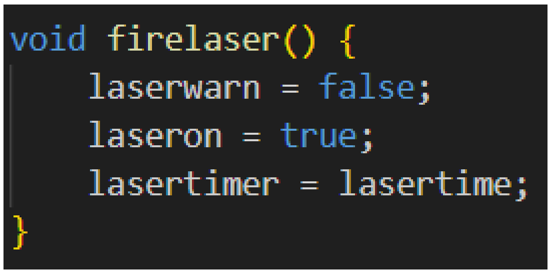
Initiates the laser warning phase by marking cells where the laser will strike. It populates **laser cells** with coordinates for horizontal and vertical lines passing through the boss, giving the player a brief window to evade before the laser activates.



```
void startlaserwarn() {
    laserwarn = true;
    laserwarntime = 7;

    lasercells.clear();
    for (int i = 1; i < width-1; i++) {
        lasercells.emplace_back(i, boss.y);
        lasercells.emplace_back(i, boss.y-1);
        lasercells.emplace_back(i, boss.y+1);
    }
    for (int i = 1; i < height-1; i++) {
        lasercells.emplace_back(boss.x, i);
        lasercells.emplace_back(boss.x-1, i);
        lasercells.emplace_back(boss.x+1, i);
    }
}
```

d) Laser Attack:

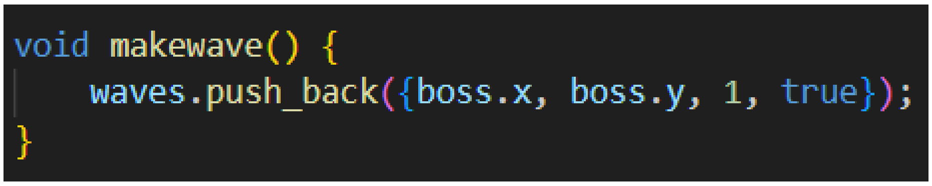
Activates the laser attack, clearing the warning state and starting the active laser phase. This function sets the **laser timer** to count down until the laser is deactivated.



```
void firelaser() {
    laserwarn = false;
    laseron = true;
    lasertimer = lasertime;
}
```

e) Make Wave:

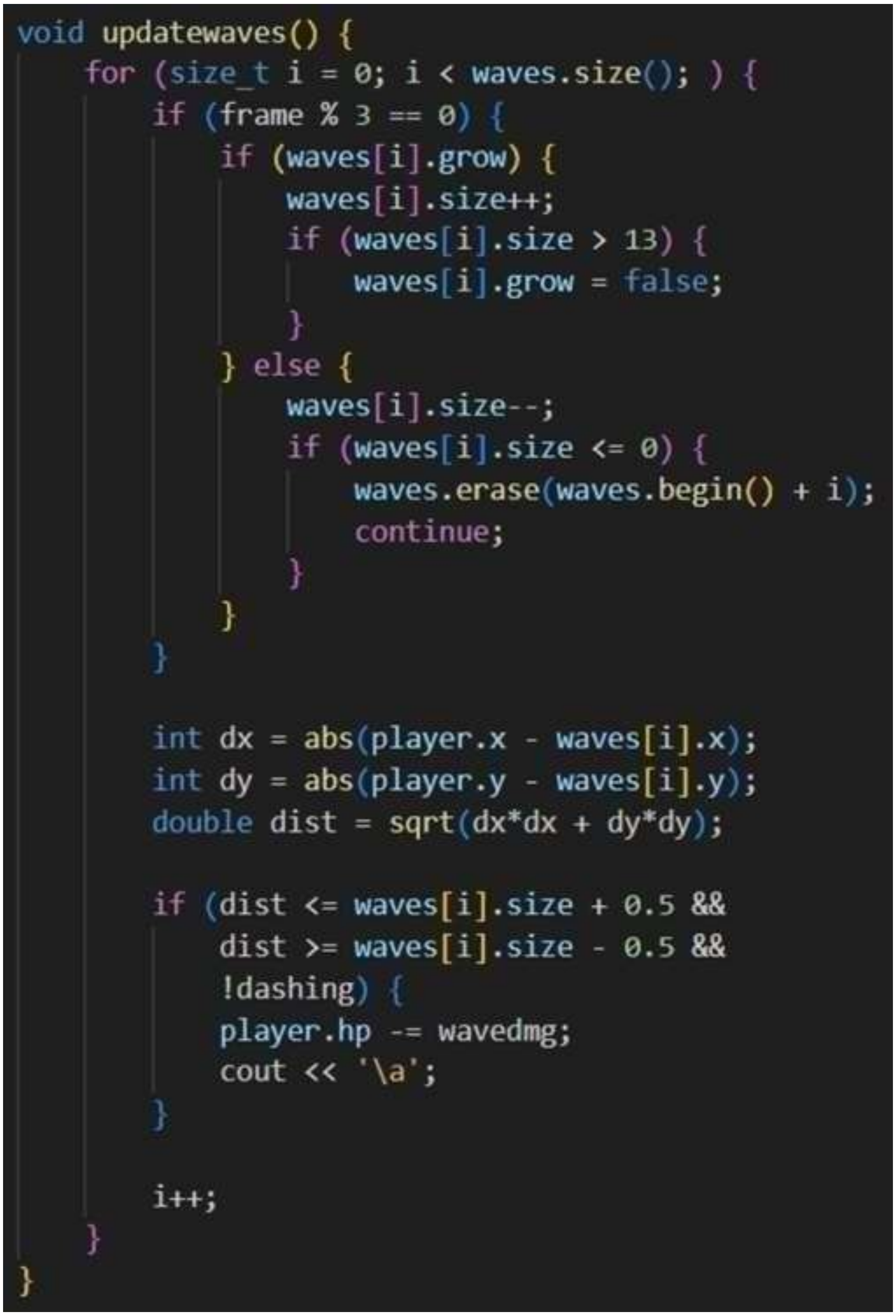
Creates a shockwave at the boss's location, adding it to the **waves** list. Each wave starts small and grows outward, eventually shrinking back and disappearing, damaging the player if they are caught in the expanding ring.



```
void makewave() {
    waves.push_back({boss.x, boss.y, 1, true});
}
```

f) Update Waves:

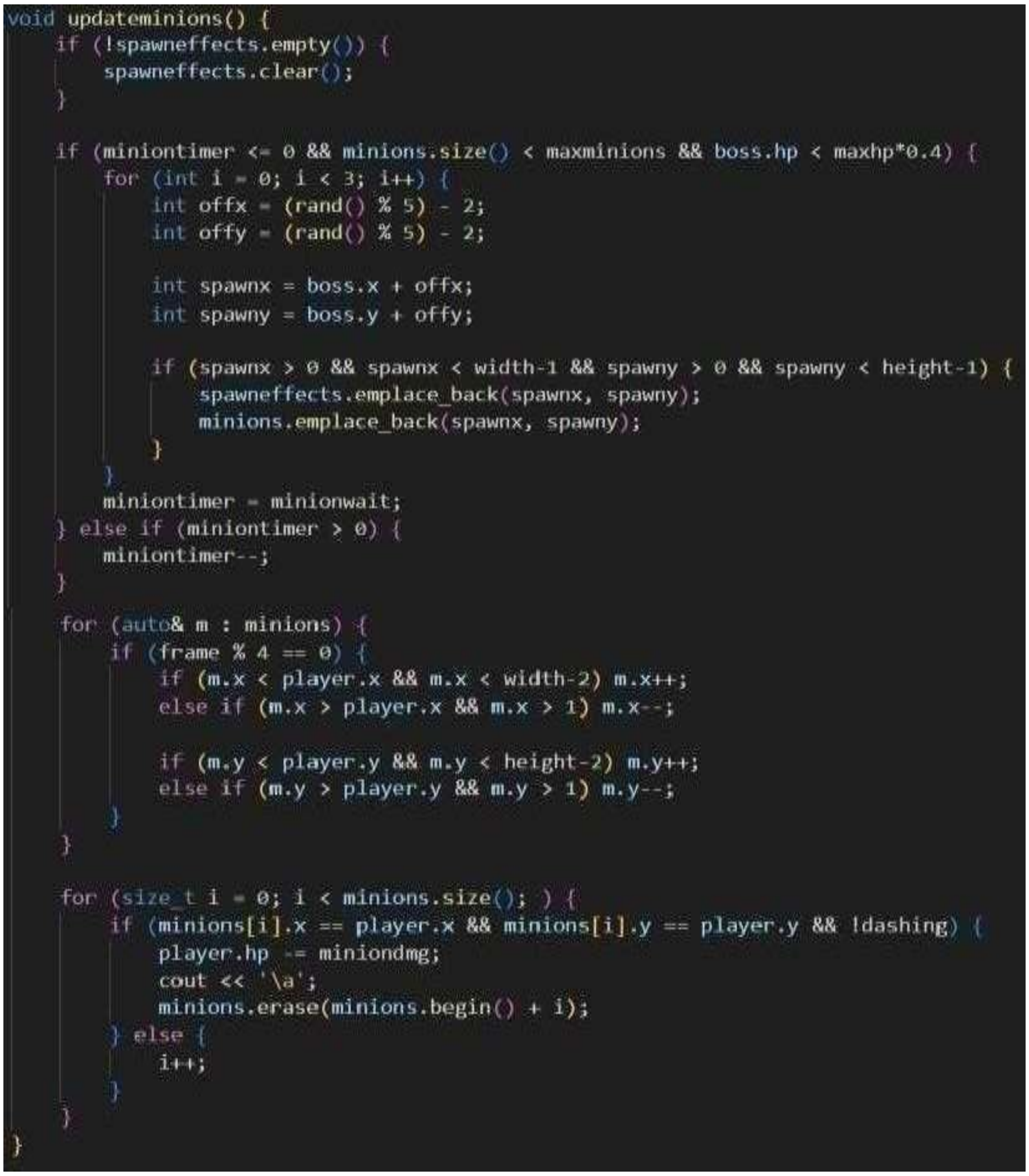
Manages the growth and shrinkage of each shockwave. It checks if the player is within the wave's effective radius, applying damage if the player is hit and removing the wave once it fully contracts.



```
void updatewaves() {
    for (size_t i = 0; i < waves.size(); ) {
        if (frame % 3 == 0) {
            if (waves[i].grow) {
                waves[i].size++;
                if (waves[i].size > 13) {
                    waves[i].grow = false;
                }
            } else {
                waves[i].size--;
                if (waves[i].size <= 0) {
                    waves.erase(waves.begin() + i);
                    continue;
                }
            }
        }

        int dx = abs(player.x - waves[i].x);
        int dy = abs(player.y - waves[i].y);
        double dist = sqrt(dx*dx + dy*dy);

        if (dist <= waves[i].size + 0.5 &&
            dist >= waves[i].size - 0.5 &&
            !dashing) {
            player.hp -= wavedmg;
            cout << '\a';
        }

        i++;
    }
}
```

g) Update Minions:

Handles the spawning and movement of minions, which appear around the boss when its health is low. It manages their approach toward the player and damage upon contact, providing additional threats alongside the boss.



```
void updateminions() {
    if (!spawneffects.empty()) {
        spawneffects.clear();
    }

    if (miniontimer <= 0 && minions.size() < maxminions && boss.hp < maxhp*0.4) {
        for (int i = 0; i < 3; i++) {
            int offx = (rand() % 5) - 2;
            int offy = (rand() % 5) - 2;

            int spawnx = boss.x + offx;
            int spawny = boss.y + offy;

            if (spawnx > 0 && spawnx < width-1 && spawny > 0 && spawny < height-1) {
                spawneffects.emplace_back(spawnx, spawny);
                minions.emplace_back(spawnx, spawny);
            }
        }
        miniontimer = minionwait;
    } else if (miniontimer > 0) {
        miniontimer--;
    }

    for (auto& m : minions) {
        if (frame % 4 == 0) {
            if (m.x < player.x && m.x < width-2) m.x++;
            else if (m.x > player.x && m.x > 1) m.x--;

            if (m.y < player.y && m.y < height-2) m.y++;
            else if (m.y > player.y && m.y > 1) m.y--;
        }
    }

    for (size_t i = 0; i < minions.size(); ) {
        if (minions[i].x == player.x && minions[i].y == player.y && !dashing) {
            player.hp -= miniondmg;
            cout << '\a';
            minions.erase(minions.begin() + i);
        } else {
            i++;
        }
    }
}
```

4.3 Structuring of game:

The **drawgame()** function is the core rendering function responsible for displaying the entire game state in the console. It begins by clearing the screen to ensure each frame is drawn freshly, avoiding overlap from previous frames. It then prints the level title based on the current stage, either "**LEVEL 1 - TRAINING**" or **"LEVEL 2 - BOSS FIGHT**", followed by the health and stamina bars for the player and boss, providing immediate feedback on their status.

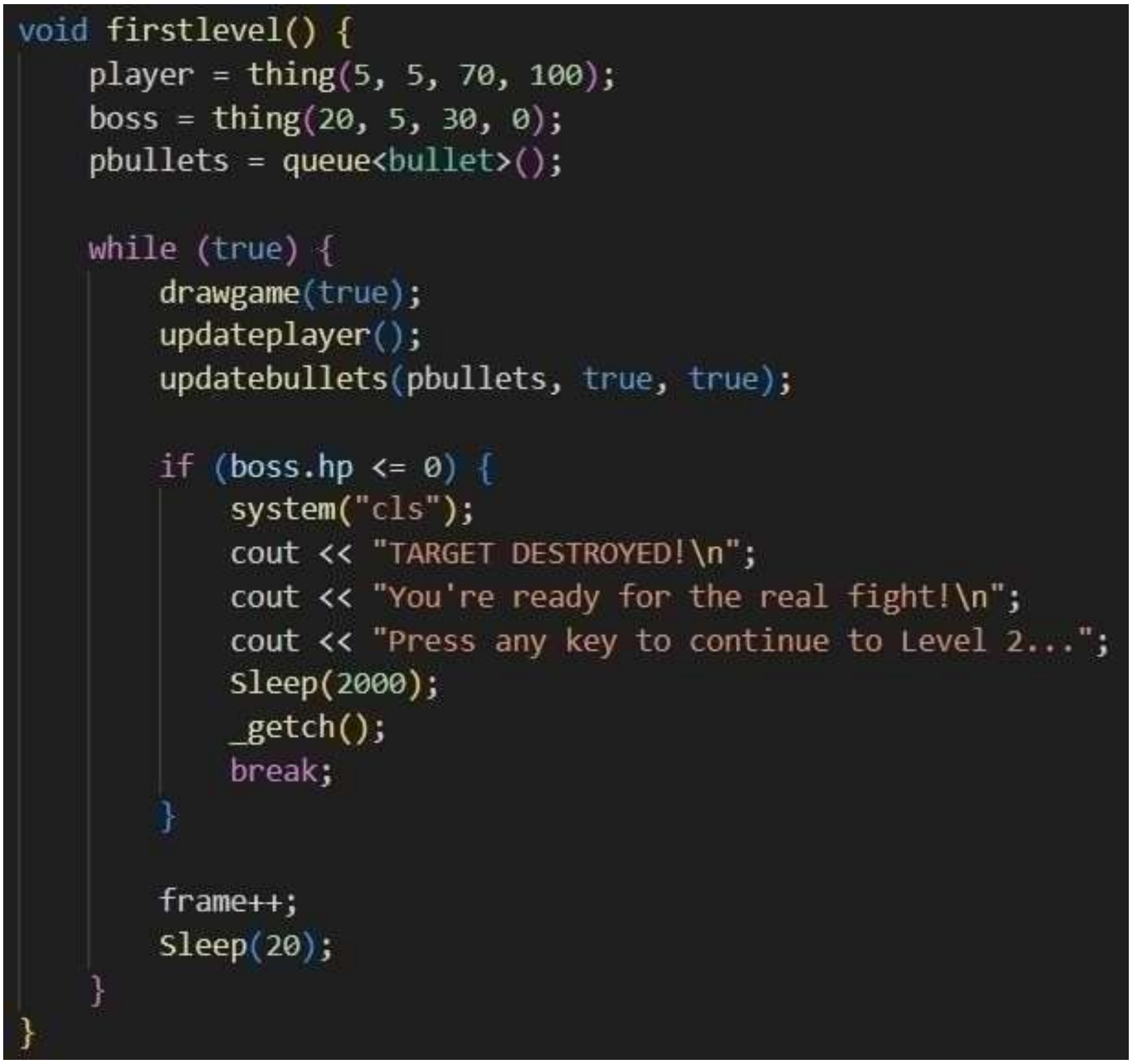
Next, it iterates over each cell in the game grid, drawing the game boundaries and placing the player, boss, bullets, minions, and special attack indicators like wave effects, laser warnings, and active lasers. The function checks for the presence of various game elements in a specific order to ensure proper layering, giving priority to objects like bullets and attacks over background elements. This approach prevents elements from being over written by later checks, maintaining a clear and accurate game display.

Once all elements are drawn, the function prints the final game frame, effectively capturing the current game state for the player to see. This tight loop of drawing and refreshing is crucial for maintaining the game's real-time, responsive feel.

The structuring of the Grid Clash game follows an object-oriented design, where each major component—such as the player, boss, bullets, and the grid—is represented through separate classes. This modular approach makes the code easier to manage, update, and debug. The main game loop handles input, logic updates, and rendering in a continuous cycle to simulate real-time gameplay. Functions are clearly divided for specific tasks like movement, collision detection, shooting, and health management. The game grid is maintained using a 2D array, allowing smooth tracking of all elements on the field. This structured layout ensures scalability and flexibility for adding new features or modifying existing ones.

a) Level -1:

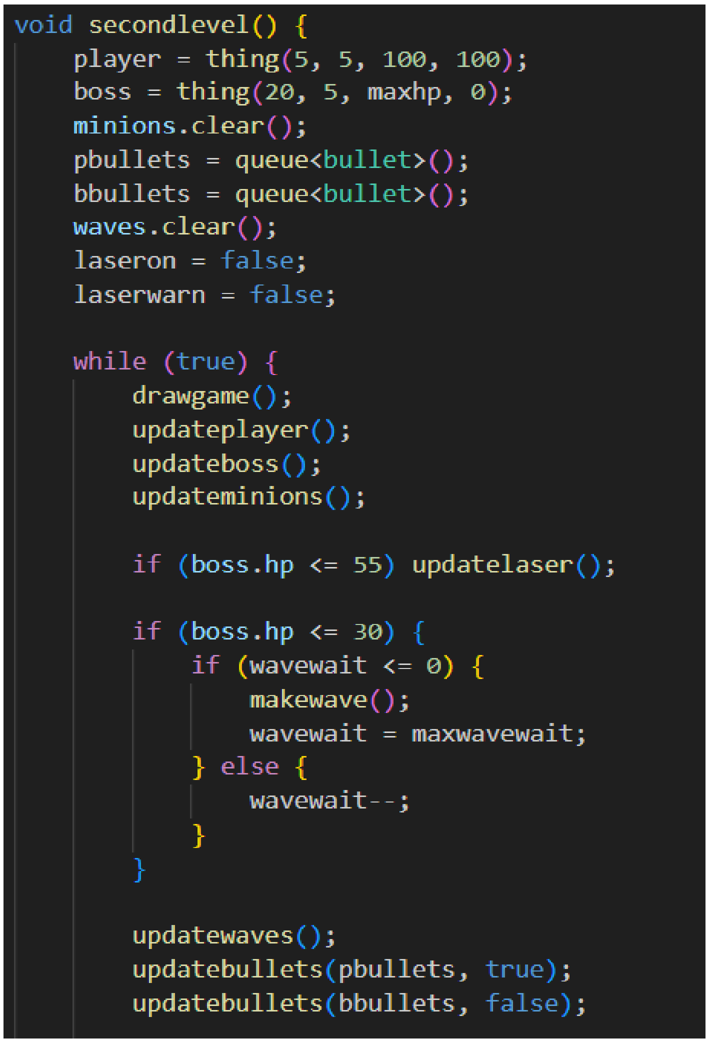
- This level is designed as a **simple introductory stage** to help players get familiar with basic controls and gameplay mechanics.
- Typically, the boss or advanced enemy behaviors are **disabled or minimal** here to keep the challenge low.
- Player focuses on navigating, avoiding simple obstacles, orbasic enemies without facing complex attacks like lasers or bullets from the boss.



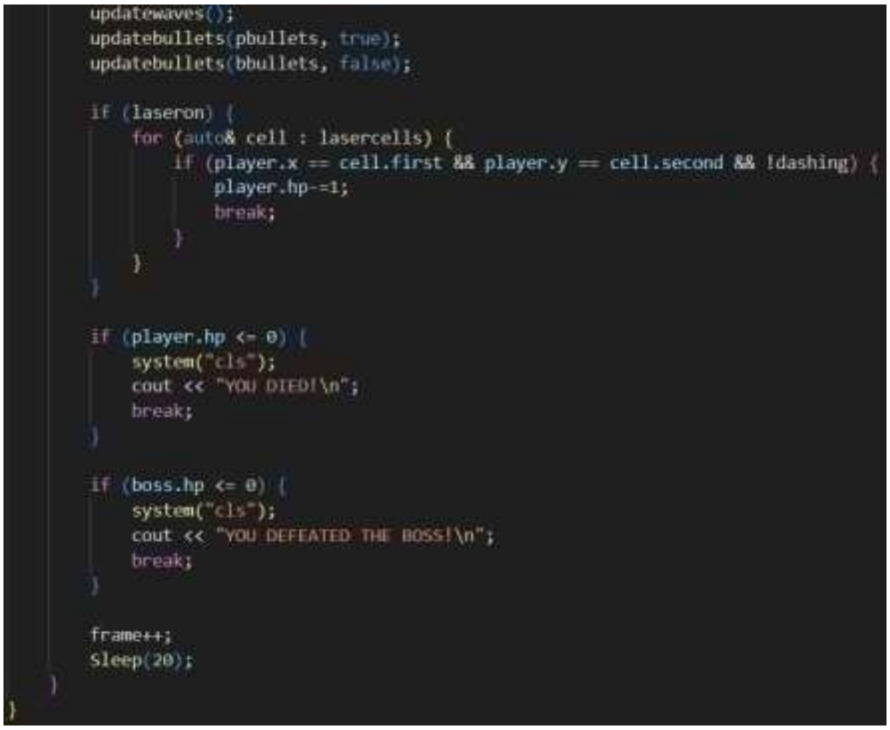
```
void firstlevel() {
    player = thing(5, 5, 70, 100);
    boss = thing(20, 5, 30, 0);
    pbullets = queue<bullet>();

    while (true) {
        drawgame(true);
        updateplayer();
        updatebullets(pbullets, true, true);

        if (boss.hp <= 0) {
            system("cls");
            cout << "TARGET DESTROYED!\n";
            cout << "You're ready for the real fight!\n";
            cout << "Press any key to continue to Level 2...";
            Sleep(2000);
            _getch();
            break;
        }

        frame++;
        Sleep(20);
    }
}
```

b) Level -2:

- The second level introduces the full boss fight mechanics with increased difficulty.
- The boss actively moves towards the player, fires bullets, and uses special attacks like lasers.
- The player must dodge more aggressive attacks requiring better strategy and skill to survive.
- This level serves as the main challenge of the game where all advanced enemy AI and attack patterns are showcased.



```cpp
void secondlevel() {
    player = thing(5, 5, 100, 100);
    boss = thing(20, 5, maxhp, 0);
    minions.clear();
    pbullets = queue<bullet>();
    bbullets = queue<bullet>();
    waves.clear();
    laseron = false;
    laserwarn = false;

    while (true) {
        drawgame();
        updateplayer();
        updateboss();
        updateminions();

        if (boss.hp <= 55) updatelaser();

        if (boss.hp <= 30) {
            if (wavewait <= 0) {
                makewave();
                wavewait = maxwavewait;
            } else {
                wavewait--;
            }
        }

        updatewaves();
        updatebullets(pbullets, true);
        updatebullets(bbullets, false);
```



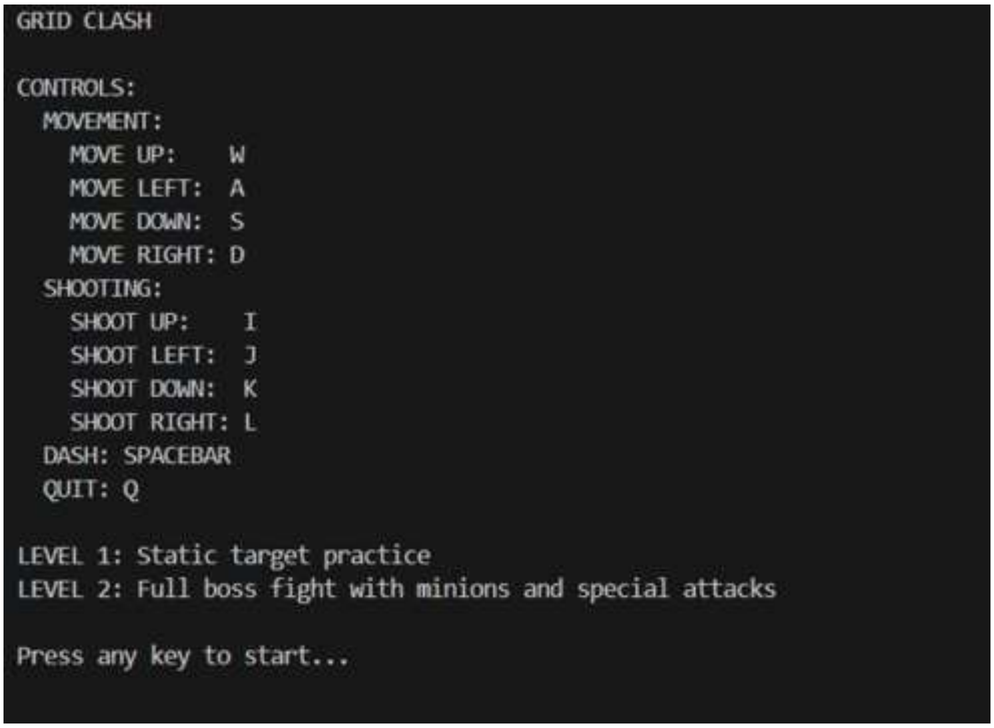
```cpp
        updatewaves();
        updatebullets(pbullets, true);
        updatebullets(bbullets, false);

        if (laseron) {
            for (auto& cell : lasercells) {
                if (player.x == cell.first && player.y == cell.second && !dashing) {
                    player.hp-=1;
                    break;
                }
            }
        }

        if (player.hp <= 0) {
            system("cls");
            cout << "YOU DIED!\n";
            break;
        }

        if (boss.hp <= 0) {
            system("cls");
            cout << "YOU DEFEATED THE BOSS!\n";
            break;
        }

        frame++;
        Sleep(20);
    }
}
```

5. RESULTSAND ANALYSIS

The game successfully demonstrates the integration of data structures such as arrays and queues to manage gameplay elements like map updates, bullet tracking, and movement. During testing, the transition between levels, player control responsiveness, and boss behavior functioned as expected. The boss's attacks, including the vertical line targeting and bullet firing, were accurately triggered based on proximity and timing conditions. Player health and stamina were updated correctly, and edge cases like boundary collisions and bullet hits were handled smoothly. The game logic remained consistent across multiple playthroughs, confirming its stability. Overall, the project met its design goals and effectively showcased both gameplay mechanics and data structure applications.

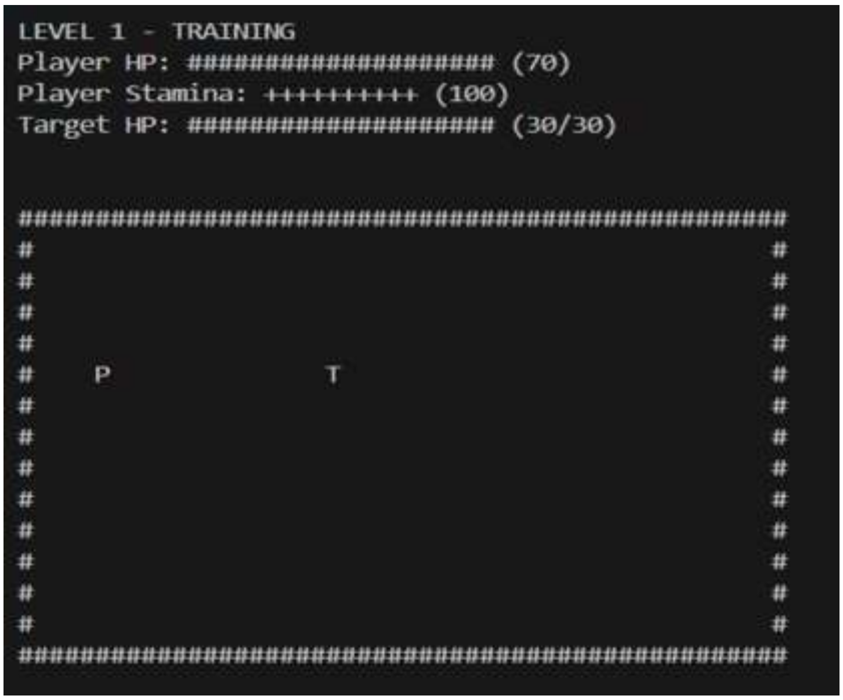
5.1 Output Screenshots:

a) Instructions: At the start of the game, clear instructions are displayed on the screen to guide the player. These include the controls for movement using the W, A, S, D keys and the key to shoot bullets using the spacebar, ensuring the player understands how to interact with the game before it begins.



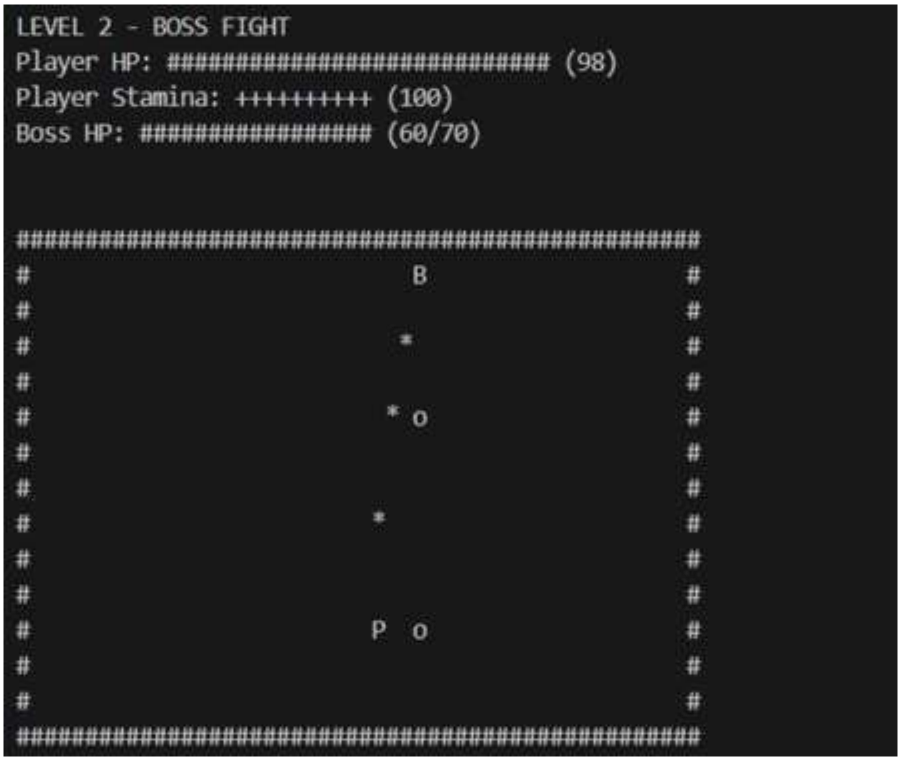
```
GRID CLASH

CONTROLS:
  MOVEMENT:
    MOVE UP:    W
    MOVE LEFT:  A
    MOVE DOWN:  S
    MOVE RIGHT: D
  SHOOTING:
    SHOOT UP:    I
    SHOOT LEFT:  J
    SHOOT DOWN:  K
    SHOOT RIGHT: L
  DASH: SPACEBAR
  QUIT: Q

LEVEL 1: Static target practice
LEVEL 2: Full boss fight with minions and special attacks

Press any key to start...
```

b) First level: The first level of "Grid Clash" is a simple 2D game where the player shoots bullets to defeat a randomly moving boss on a grid.



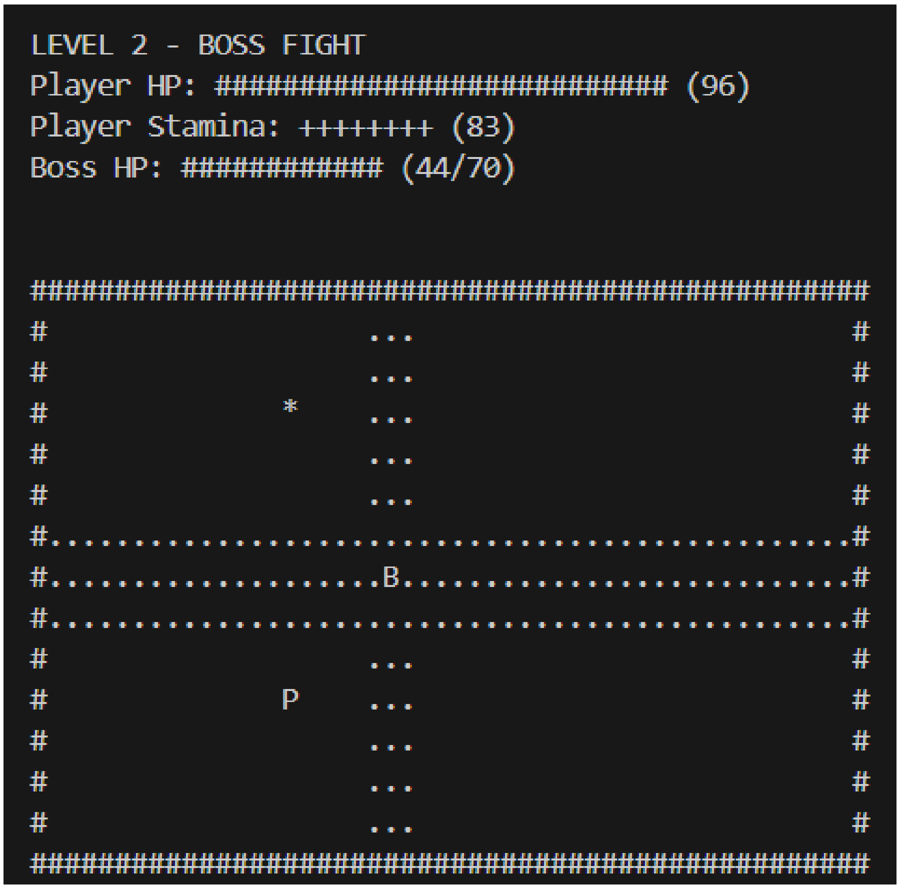
```
LEVEL 1 - TRAINING
Player HP: #################### (70)
Player Stamina: ++++++++++ (100)
Target HP: #################### (30/30)


################################################
#                                              #
#                                              #
#                                              #
#                                              #
#    P              T                          #
#                                              #
#                                              #
#                                              #
#                                              #
#                                              #
#                                              #
#                                              #
#                                              #
################################################
```

c) Second Level: In the second level of **Grid Clash**, the boss becomes more powerful with a special attack: it marks a vertical line in red and then shoots bullets down that path. The player must dodge the warning line to avoid getting hit while continuing to attack and reduce the boss's health.

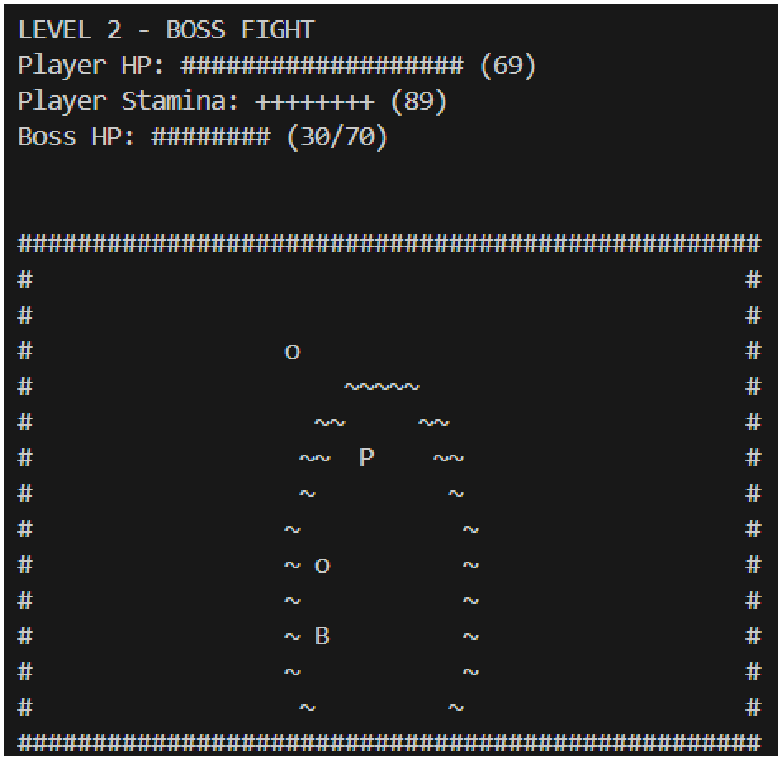


```
LEVEL 2 - BOSS FIGHT
Player HP: ############################ (98)
Player Stamina: ++++++++++ (100)
Boss HP: ################# (60/70)


##################################################
#                            B                   #
#                                                #
#                           *                    #
#                                                #
#                          * o                   #
#                                                #
#                                                #
#                         *                      #
#                                                #
#                                                #
#                         P  o                   #
#                                                #
#                                                #
##################################################
```

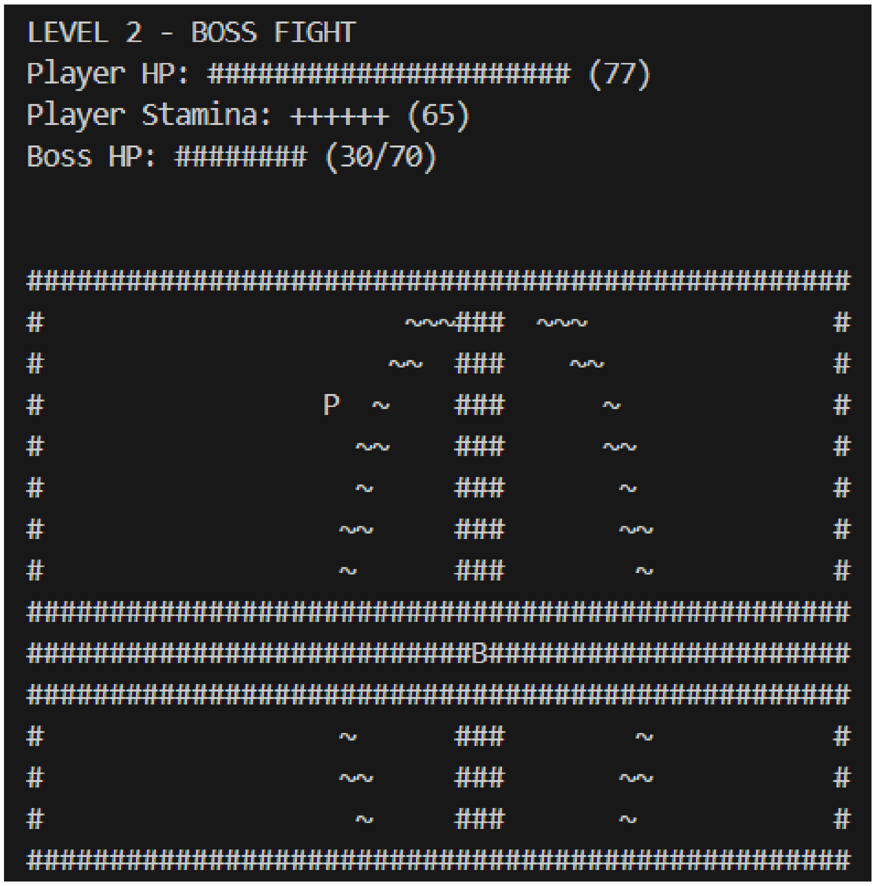
d) **Boss Attack's:** In the second level of **Grid Clash**, the boss uses multiple strategic attacks to challenge the player. The primary attack is the **Vertical Laser Strike**, where a red warning line is shown on a column, and after a delay, bullets are fired straight down that path. This tests the player's reaction time and positioning. Another attack is the **Random Shot Burst**, where the boss fires bullets in multiple directions randomly, creating chaos on the grid. Occasionally, the boss may use the **Rapid Fire**, launching a quick succession of bullets in a single column to trap the player. The attacks follow patterns that repeat with slight variations to confuse the player. The boss's **movement is random**, making its attacks unpredictable and harder to dodge. As the boss's health drops, the **attack frequency increases**, putting more pressure on the player. The player must stay alert and keep moving while timing shots to win the level.



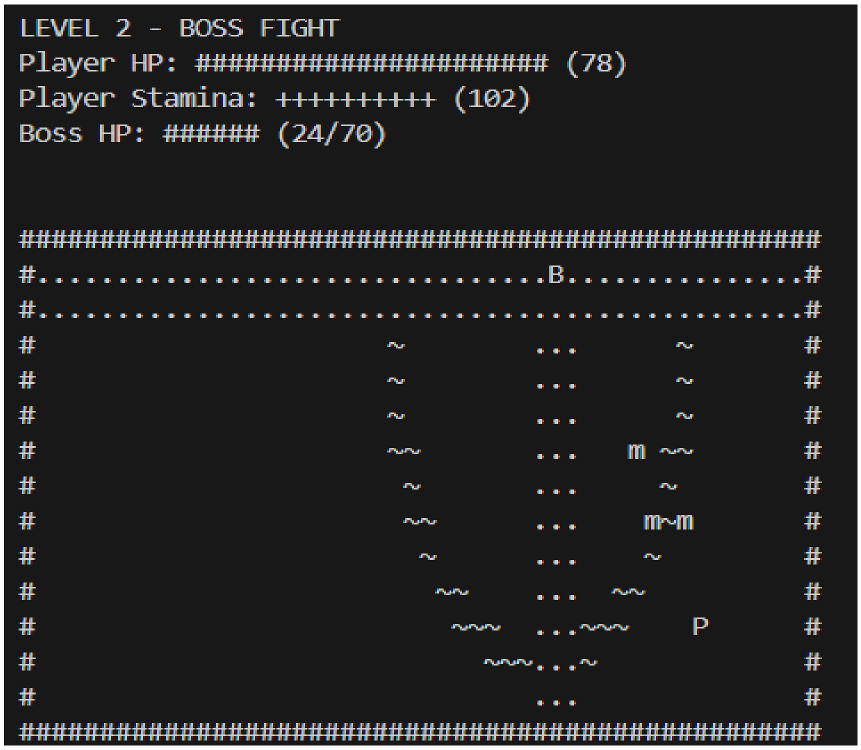
```
LEVEL 2 - BOSS FIGHT
Player HP: ########################### (96)
Player Stamina: ++++++++ (83)
Boss HP: ############ (44/70)


##################################################
#                   ...                          #
#                   ...                          #
#              *    ...                          #
#                   ...                          #
#                   ...                          #
#................................................#
#....................B...........................#
#................................................#
#                   ...                          #
#              P    ...                          #
#                   ...                          #
#                   ...                          #
#                   ...                          #
##################################################
```



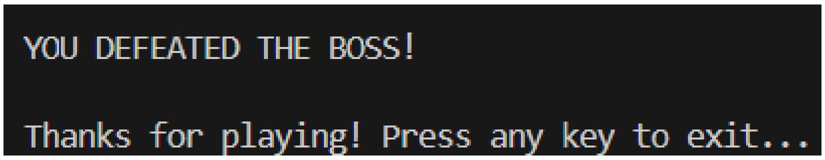
```
LEVEL 2 - BOSS FIGHT
Player HP: ################### (69)
Player Stamina: ++++++++ (89)
Boss HP: ######## (30/70)


##################################################
#                                                #
#                                                #
#                 o                              #
#                     ~~~~~                      #
#                   ~~     ~~                    #
#                  ~~  P    ~~                   #
#                  ~         ~                   #
#                 ~           ~                  #
#                 ~ o         ~                  #
#                 ~           ~                  #
#                 ~ B         ~                  #
#                 ~           ~                  #
#                  ~         ~                   #
##################################################
```



```
LEVEL 2 - BOSS FIGHT
Player HP: ###################### (77)
Player Stamina: ++++++ (65)
Boss HP: ######## (30/70)


##################################################
#                      ~~~###  ~~~               #
#                     ~~  ###    ~~              #
#                 P  ~    ###      ~             #
#                   ~~    ###      ~~            #
#                   ~     ###       ~            #
#                  ~~     ###       ~~           #
#                  ~      ###        ~           #
##################################################
###########################B######################
##################################################
#                  ~      ###        ~           #
#                  ~~     ###       ~~           #
#                   ~     ###       ~            #
##################################################
```



```
LEVEL 2 - BOSS FIGHT
Player HP: ###################### (78)
Player Stamina: ++++++++++ (102)
Boss HP: ###### (24/70)


###################################################
#................................B................#
#.................................................#
#                      ~        ...      ~        #
#                      ~        ...      ~        #
#                      ~        ...      ~        #
#                      ~~       ...   m ~~        #
#                       ~       ...     ~         #
#                       ~~      ...    m~m        #
#                        ~      ...    ~          #
#                         ~~    ...  ~~           #
#                          ~~~  ...~~~    P       #
#                            ~~~...~              #
#                               ...               #
###################################################
```



```
YOU DEFEATED THE BOSS!

Thanks for playing! Press any key to exit...
```

5.2 Time Complexity:

The time complexity of this code is mainly influenced by the game loop, which repeats every frame. drawgame() takes O(H × W) as it refreshes the entire grid, checking each cell individually. Player and boss updates are generally O(1), involving basic position adjustments. Bullet handling is O(B), as each bullet is checked for movement and collisions. For Level 2, the addition of minions (O(M)) and waves (O(W)) further increases the complexity. Overall, the code is roughly O(H × W + B + M + W) per frame, depending on the number of active elements.

5.3 Space Complexity:

The space complexity of this code depends on the storage required for the game grid, bullets, minions, waves, and laser cells. The grid itself is not explicitly stored as a 2D array, which saves space, but dynamic structures like **pbullets**, **bbullets**, **minions**, **waves**, and **lasercells** consume additional memory. These structures scale with the number of active elements, making the space complexity roughly **O(B + M + W + L)**, where **B** is the total number of bullets, **M** is the number of minions, **W** is the number of active waves, and **L** is the number of laser cells. The fixed-size objects like **player** and **boss** are **O(1)**, while the vector sizes for minions and waves depend on the game's progress, leading to potentially high memory usage if many elements are active simultaneously.

6. CONCLUSION

This project, a 2D grid-based action game, demonstrates the integration of data structures and algorithms into an interactive gameplay experience. By carefully managing player controls, boss behavior, bullet interactions, and wave mechanics, it highlights the importance of efficient coding and real-time decision-making. The use of queues for bullet management, vector-based minion tracking, and structured class-based design allowed for a balanced and challenging gameplay. Additionally, the project incorporated concepts like hit detection, stamina management, and special attack patterns, reflecting a comprehensive approach to game development. Overall, it effectively bridges the gap between theoretical data structure concepts and practical game design, providing a solid foundation for further exploration in game programming. Building on this foundation, the project also emphasizes modular code organization and scalability, making it easier to expand with new features or mechanics. The real-time application of algorithms like BFS for enemy pathfinding or cooldown timers for abilities showcases how academic concepts translate into dynamic gameplay. This blend of theory and creativity not only enhances the game's educational value but also prepares developers for more complex game development challenges.

7. FUTURE WORK

The future of the **Grid Clash 2D** game holds tremendous potential as it can evolve into a more immersive and feature-rich experience through both technical and gameplay enhancements. By incorporating **advanced AI algorithms**, the game could introduce enemies with more realistic and unpredictable behavior, such as pathfinding, teamwork among enemies, or reactive strategies based on the player's actions. Enhanced **boss mechanics**—including multi-phase battles, randomized attack patterns, and adaptive difficulty scaling—would make boss fights significantly more thrilling and less predictable.

Future iterations could also benefit from **dynamic grid layouts** that evolve in real time, introducing destructible environments, shifting tiles, or environmental hazards like traps or power zones. These additions would force players to constantly adapt their strategies, adding an extra layer of depth and replay ability.

From a gameplay perspective, **multiplayer support**, whether local or online, could transform Grid Clash 2D from a single-player experience into a **competitive or cooperative game**, fostering a strong player community. Adding **online leaderboards**, seasonal challenges, and player rankings would further boost replay value and encourage continued engagement.

From a technical and user experience standpoint, **implementing file handling for persistent game states** would allow players to save and resume their progress, enabling longer and more complex campaigns. Support for **cross-platform compatibility**, including mobile versions, tablets, or even web-based play, could significantly increase the accessibility and reach of the game. Enhanced **visuals, animations, and audio effects** would improve immersion and attract a broader audience.

Overall, with a clear development roadmap and community-driven innovation, Grid Clash 2D has the potential to transition from a simple 2D grid battle game into a **strategically deep, technically advanced, and widely loved** gaming experience.

8. REFERENCES

1. Game Programming in C++: Creating 3D Game: https://books.google.com/books?hl=en&lr=&id=VfxNDwAAQBAJ&oi=fnd&pg=PT24&dq=references+related+to+game+projects+in+c%2B%2B+using+data+structures+&ots=Dnaao-N6ue&sig=TqHwuq3-vbqPVR7VZQHebS9-lys
2. Learn C++ for game development: https://books.google.com/books?hl=en&lr=&id=XAfXAwAAQBAJ&oi=fnd&pg=PP3&dq=references+related+to+game+projects+in+c%2B%2B+using+data+structures+&ots=NrVlNVCgD0&sig=nDu4eOHTsfzOA0M1M1M7VHrmUPc
3. Impact of algorithmic and data structure implementation to game development: https://www.ewadirect.com/proceedings/ace/article/view/9885
4. The Game in C++: https://skemman.is/handle/1946/3207
5. Beginning C++ Game Programming: Learn C++ from scratch by building fun games: https://books.google.com/books?hl=en&lr=&id=53QKEQAAQBAJ&oi=fnd&pg=PP1&dq=header+files+in+c%2B%2B+for+game+structuring+2d+github&ots=-3fPRoBvnL&sig=7geshBd7kuA6mc8p9MOjXEUmd1U