Programming and Fundamental Algorithms Report

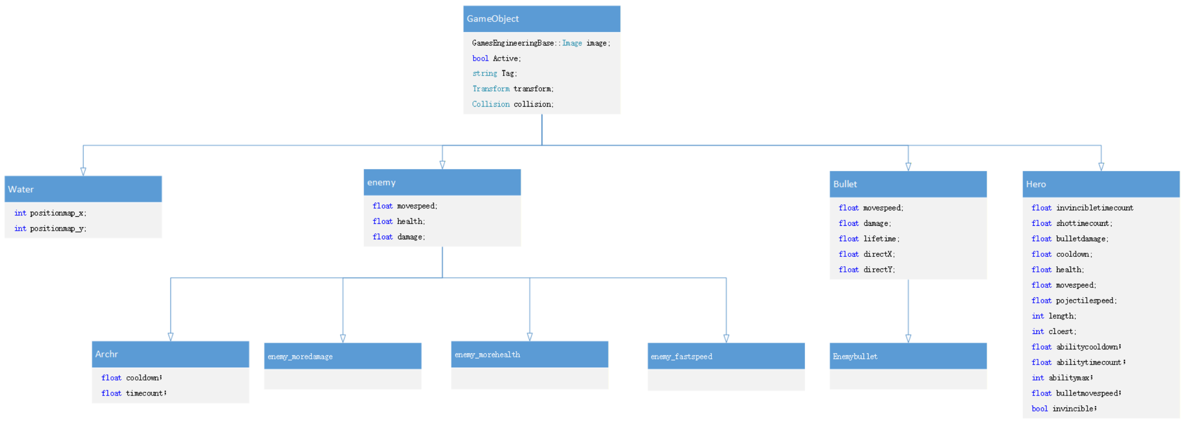
Jiachen Lyu

November 9, 2025

Contents

1 Introduction

This project is a vampire survivor-like game based on GamesEngineeringBase, which is implemented using multiple C++ techniques learned in the course. Each game lasts two minutes and offers a variety of levels to choose from. It also features different enemies and items to enhance the hero characters. The project takes visual studio as the development platform, uses various technologies such as classes, objects, Pointers, and virtual functions in C++, and introduces an external JSON library for data saving and loading. Here we briefly introduce some important classes and their inheritance relationships. All interactive units in the game inherit from the GameObject class, which provides some of the most basic variables for units such as labels, positions, colliders, and activation statuses. Among them, the position and collider are also separate classes, used to record the position of units and the position and size of colliders. He has four important subclasses: the hero class, the enemy class, the Water class, and the Bullet class. Among them, the hero class adopts the singleton pattern to create and record the Instance of the hero character. The enemy class has four subclasses, each representing a different type of enemy. The Bullet class is used to record the bullets fired by heroes. It also has a subclass, Enemybullet, which is used to record bullets fired by enemies. The Water class is used to record the water in the background to complete the restriction on the hero character. There are three classes for managing objects in the game, namely the GameObjectManager class, the enemiesmanager class, and the Bulletmanager class. Among them, the GameObjectManager class is mainly used to determine whether units collide, while the enemiesmanager class is used to generate enemies and update their positions for each frame. The Bulletmanager class mainly updates the position of the enemy's bullets every frame. All three management classes adopt the singleton pattern. The GameObjectManager instance has a secondary pointer pointing to GameObject, which stores the information of all GameObject instances and dynamically expands based on the number of instances. The enemiesmanager class stores information about all instances of the enemy class and its subclasses, while the Bulletmanager class stores information about all instances of the Bullet class and its subclasses. The storage method is similar to that of the GameObjectManager class. Instances of each class will be automatically registered when generated and removed from the management class when destroyed.


```
GameObject
GamesEngineeringBase::Image image;
bool Active;
string Tag;
Transform transform;
Collision collision;

Water
int positionmap_x;
int positionmap_y;

enemy
float movespeed;
float health;
float damage;

Bullet
float movespeed;
float damage;
float lifetime;
float directX;
float directY;

Hero
float invincibletimecount
float shottimecount;
float bulletdamage;
float cooldown;
float health;
float movespeed;
float pojectilespeed;
int length;
int cloest;
float abilitycooldown;
float abilitytimecount;
int abilitymax;
float bulletmovespeed;
bool invincible;

Archr
float cooldown;
float timecount;

enemy_moredamage

enemy_morehealth

enemy_fastspeed

Enemybullet
```

Figure 1: Main Class Structure Relationships

2 Technical Implementation

2.1 Virtual Camera

To implement a virtual camera, this project introduces a Camera class that defines X and Y offset values. During rendering, all backgrounds and game objects subtract these offset values from their world positions to ensure that the hero always remains centered on the screen, while other entities and backgrounds maintain the correct relative positions around the hero. The project uses the top-left corner of the map as the origin point (0, 0). At the start of the game, the hero's world coordinates are set to (640, 640) — the center of a 1280×1280 window. Therefore, the camera's X and Y offsets are initially set to subtract 640, ensuring that the offset starts at 0 and provides accurate relative movement afterward. All units and background perform their logical updates based on their world coordinates, but when drawn, their positions are adjusted by subtracting the camera's offset values.

For example, if an object's world X coordinate is 2560 and the camera's X offset is 1480, its window coordinate becomes 1080, which represents its actual rendered position on the window. In implementation, the Camera class defines two private float variables, offsetX and offsetY, along with corresponding get and set methods. The class follows the Singleton pattern — its constructor is private, and it provides a public static function that returns a reference to the camera instance. This static local variable is created only once when first accessed, allowing convenient and consistent access to the same camera reference throughout the program.

Listing 1: Get Camera Instance Function

```
static Camera& GetCamera()
 {
     static Camera camera;
     return camera;
 }
```

In general, during the game's lifecycle, whenever the hero's position changes, the camera's offset values are updated to match the hero's current world coordinates, while subtracting the hero's initial position (640, 640) as a correction factor.

Listing 2: Initialize Camera Position

```
camera.SetPosition(hero.transform.GetPositionX()
- (canvas.getWidth() / 2), hero.transform.GetPositionY()
- (canvas.getHeight() / 2));
//set position while subtracting the hero's initial position
```

When rendering the window, the program first calculates the world coordinate position corresponding to each pixel based on the world coordinates of the background and game objects. It then subtracts the camera's offset values from these coordinates before drawing them on the window.

Listing 3: Window Coordinate Drawing Method

```
//correction position with camera offset
void camera_draw(float x, float y, GamesEngineeringBase::Image& image,
        GamesEngineeringBase::Window& canvas, GameObject* obj) {

        if (obj->transform.GetPositionX() + x - Camera::GetCamera().GetX() >= 0
                && obj->transform.GetPositionX() + x - Camera::GetCamera().GetX() < canvas.getWidth()
                && obj->transform.GetPositionY() + y - Camera::GetCamera().GetY() >= 0 &&
                obj->transform.GetPositionY() + y - Camera::GetCamera().GetY() < canvas.getHeight())
                //if the new position in the canvas
                canvas.draw(obj->transform.GetPositionX() + x
                        - Camera::GetCamera().GetX(),
                        obj->transform.GetPositionY() + y
                        - Camera::GetCamera().GetY(), obj->image.at(x, y));
}
```

When the map boundaries are fixed, the camera's offset values are constrained within a certain range to ensure that the camera's view never moves outside the game area.

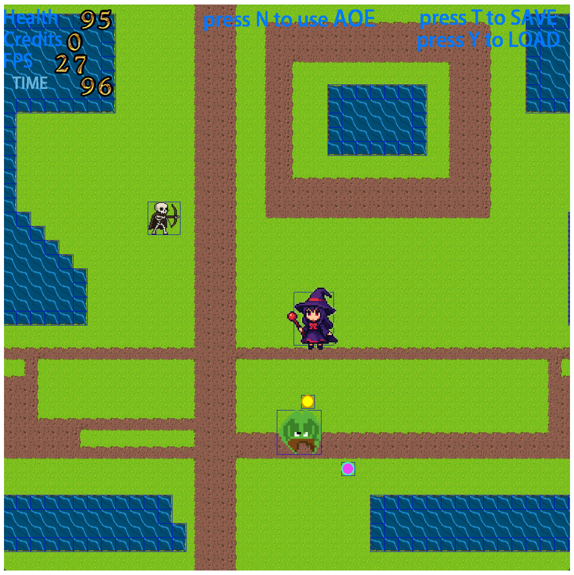


Figure 2: Camera with Hero in First Position

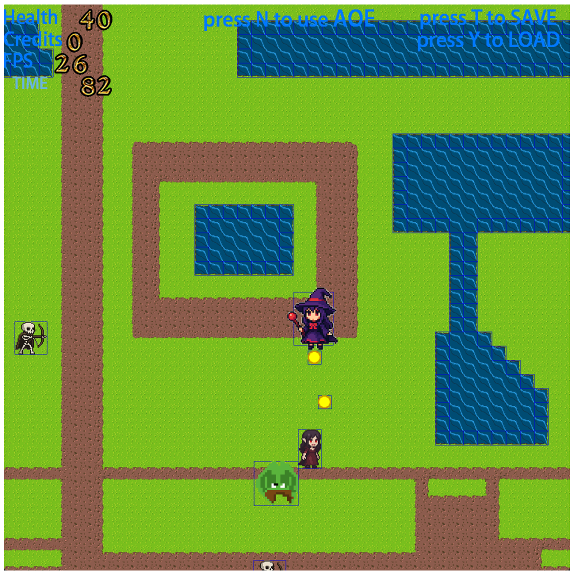


Figure 3: Camera with Hero in Second Position

2.2 NPCs System

2.2.1 Generate Randomly in Outside

To achieve timed enemy spawning and efficient enemy management, this project implements an enemiesmanager class. After a set time interval, it randomly generates an enemy of a certain type. By randomly determining both the distance from the hero and a unit direction vector, the system ensures that enemies are spawned within an annular area around the hero. Since the camera is always centered on the hero, this design guarantees that enemies will not appear within the camera's visible area upon spawning. The spawning process first retrieves the hero's current position. Then, random values between 1500 and 2000 are added to the hero's X and Y coordinates, and a random value between -1 and 1 is generated as the unit vector's X component. The corresponding Y component is then calculated based on this value. Multiplying the direction vector by the chosen distance makes the final spawn position, meaning that enemies will appear within a annular zone 1500 to 2000 units away from the hero.

Listing 4: Function about Creat New archr Instance

```
void enemiesmanager::createnemyarchr(float x, float y,
    float speed, int health, int damage) {
    getrandom(x, y);
    enemy* scv3 = new Archr(x, y);
}
```

Listing 5: Calculate Random Position Method

```
float getrandom(float &a,float &b) {
    //Directly modify the given x and y coordinates
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> distInt(0, 99);
    uniform_real_distribution<float>
        distFloat(a+1500.f, a+2000.f);
    uniform_real_distribution<float>
        distFloat1(-1.f, 1.f);
    float x = distFloat1(gen);
    float y = sqrt(1 - (x * x));
    //Calculate the unit vector of y based on x
    a = x * distFloat(gen);
    b = y * distFloat(gen);
```

```
        return 0;
}
```

2.2.2 Enemies Frequency Increases Over Time

After the game starts, the elapsed time is recorded, and the enemy spawn interval is gradually reduced as time progresses. In the enemiesmmanager class, the changecodwon method is executed once per frame to track the elapsed time. Every 25 seconds, the variable that records the total elapsed time since the game began is reset to zero, and the variable controlling the enemy spawn interval is decreased by 0.3 seconds. This process ensures that the spawn frequency increases over time, making the game progressively more challenging.

Listing 6: Update Time Function

```
//reduce enemy spawn interval with time flow
void changecodwon(float cd) {
    timecounttotal += cd;
    if (timecounttotal >= 25.f) {
        timecounttotal -= 25.f;
                        cooldown -= 0.3f;
    }
}
```

2.2.3 Four Different Character Types

In this project, the enemy class, which inherits from the GameObject class, serves as the base class for all enemy types. Four subclasses are derived from the Enemy class to create different kinds of enemies. When constructing each subclass, different health and speed values are assigned, and the corresponding picture is loaded. During gameplay, the enemiesmmanager sequentially creates instances of these four subclasses to generate a variety of enemies.

Listing 7: Enemy with More Damage Constructor

```
/constructing enemy with more damage
enemy_moredamage(float x, float y) :enemy(x, y, 50.f, 100, 10.f) {
        //set the enemy positionX,positionY,movespeed,helath,damage
        image.load("../Resources/xixuegui.png");
    //set new picture for NPC
        collision.SetCollision(transform.GetPositionX(),
                transform.GetPositionY(), image.width, image.height);
}
```

2.2.4 NPCs Behavior

After obtaining the hero's position, the program calculates the unit vector from each NPCs to the hero. Then, in every frame, each NPC moves toward the hero according to its movement speed. In the enemiesmmanager class, the updateAll() method is executed once per frame. It is responsible not only for spawning enemies but also for updating the movement of all NPCs. The method first calculates the time elapsed since the previous frame, multiplies it by each NPC's movement speed, and then computes the direction vector from the NPC to the hero. The resulting displacement is added to the NPC's current position to determine its new position. Finally, the NPC's position and collision box are updated accordingly.

Listing 8: NPCs Moving Function

```
//calculate the new position
if (enemies[i]->Tag != "Archr")
enemies[i]->transform.SetPosition(
    enemies[i]->transform.GetPositionX() +
    ( getDirectionX(enemies[i], &Hero::getInstance())
        *dt*enemies[i]->getmovespeed()),
    enemies[i]->transform.GetPositionY() +
    (getDirectionY(enemies[i], &Hero::getInstance()) * dt
        * enemies[i]->getmovespeed()));
//the special rule for archer,it will move when hero beyond its range.
else if(enemies[i]->Tag == "Archr") {
    if (getDistance(enemies[i], &Hero::getInstance()) >= 800) {
        enemies[i]->transform.SetPosition
        (enemies[i]->transform.GetPositionX() +
            (getDirectionX(enemies[i], &Hero::getInstance())
                * dt * enemies[i]->getmovespeed()),
            enemies[i]->transform.GetPositionY() +
            (getDirectionY(enemies[i], &Hero::getInstance())
                * dt * enemies[i]->getmovespeed()));
    }
    else {
        enemies[i]->updatetime(dt);
        //update archer shot interval
    }
}
enemies[i]->Update();
//update collision box
```

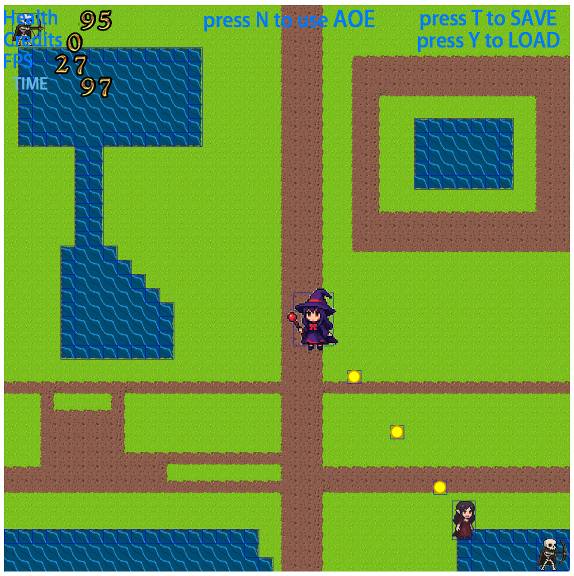


Figure 4: NPC Position Before Movement

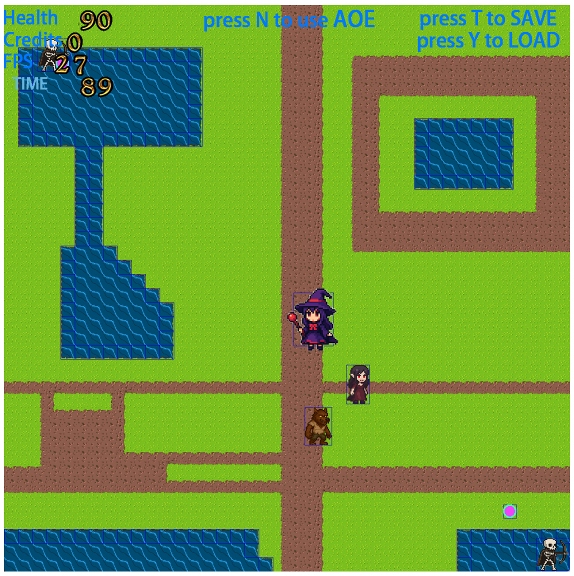


Figure 5: NPC Position After Movement

2.2.5 Archer

The Archr class inherits from the enemy class. During NPC movement updates in the enemiesmanager class, the program checks whether an NPC is an instance of Archr. If it is, the Archr is excluded from movement updates and instead has its internal timer updated, ensuring that it can periodically fire projectiles. The Enemybullet class inherits from the Bullet class and represents projectiles fired by enemies. It is also managed by BulletManager. To prevent projectiles from persisting indefinitely, the Bulletmanager updates the lifetime of all Bullet instances and their subclasses. Once the remaining time of a bullet reaches zero, it is destroyed—effectively providing a range limitation for all projectiles. Within the Archr class, the Tag is changed from "enemy" to "Archr", while the other three enemy subclasses retain the "enemy" tag. Archr can create new Enemybullet instances using the fire() method, which sets the bullet's initial position to the Archr's current position and calculates the direction vector towards the hero. The updateTime(float dt) function updates the Archr's internal timer to control when it fires bullets; this function overrides a virtual function from the enemy class. The Enemybullet class, inheriting from Bullet, also changes its Tag to "Enemybullet" and loads a different picture.

During position updates in the enemiesmmanager, the system calculates the distance between an Archr and the hero. If the distance is within 800 units, the Archr stops moving to maintain its firing position.

Listing 9: archr Class Constructor

```
Archr(float x,float y) :enemy(x,x, 50, 100, 0) {

                Tag = "Archr";
                image.load("../Resources/archr.png");
                collision.SetCollision(transform.GetPositionX(),
transform.GetPositionY(),image.width,image.height);

        }
```

Listing 10: Create Enemybullet Instance Function

```
        void fire(float dt) {

        timecount += dt;
        if(timecount>=cooldown) {
                timecount = 0;
                Enemybullet* bullet = new Enemybullet();
                //create new Enemybullet instance
                bullet->transform.SetPosition
                (transform.GetPositionX(),transform.GetPositionY());
                //set bullet position as archer position
                bullet->setdirection(getDirectionX
                (this, &Hero::getInstance()), getDirectionY(this,
                  &Hero::getInstance()));
                //set direction
        }

}
```

2.3 Collision System

This project implements a collision system using rectangular bounding boxes. A dedicated Collision class is defined and referenced by each GameObject, allowing all game instance to possess a collision box. During each update cycle, the game runs a nested loop to check for collisions between every pair of objects, thereby completing the collision detection process. The Collision class contains six float variables and one boolean variable. The float variables store the X and Y positions, width, height, and offsets in both the X and Y directions. The bounding box's width and height are typically defined when the object is created—usually matching the width and height of the object's sprite. The collision box position always matches the object's position and is updated every frame after the object's position changes. If the collision box size is not explicitly defined during creation, its width and height default to 0. The offset values determine the collision box's position relative to the object's position. For example, if the X offset is 10, the collision box's X position will be the object's X position plus 10. The collision detection uses the Axis-Aligned Bounding Box method. Each frame, the GameObjectManager performs a Nested Loop to check for overlaps between all active objects. For testing and debugging purposes, the draw collision method can render each collision box based on its position, width, and height, making it easier to visualize and verify collisions.

Listing 11: Axis-Aligned Bounding Box Method

```
bool isColliding(Collision& a,Collision& b) {
        return !(a.x + a.w < b.x ||  b.x + b.w < a.x
    ||  a.y + a.h < b.y ||  b.y + b.h < a.y);
}
```

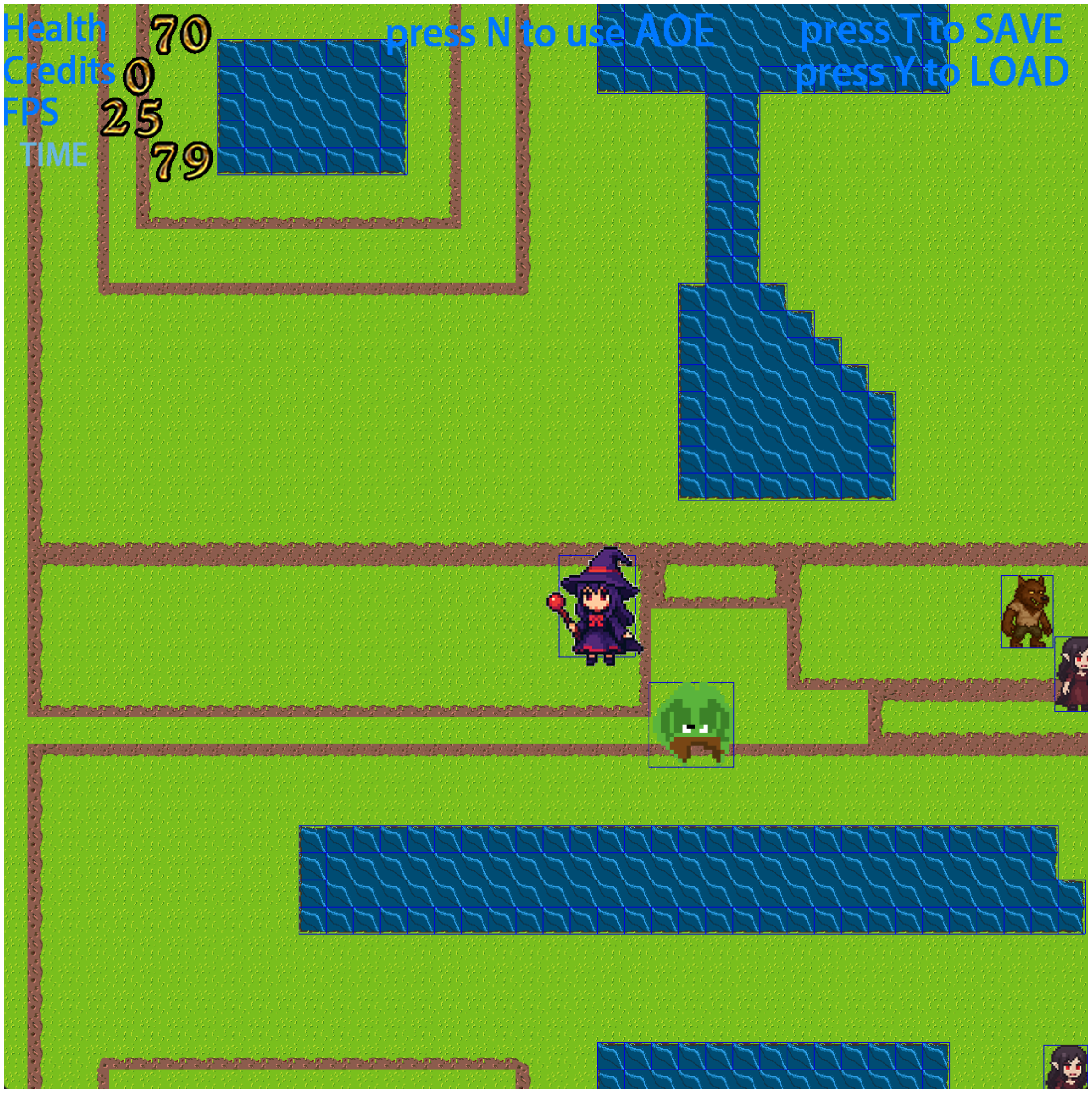


Figure 6: The Draw about Collision Box

2.3.1 Hero vs NPCs

In the nested loop, if two objects are detected with Tags of "Hero" and "Enemy", the system registers a collision event in which the hero takes damage from that NPC.

Listing 12: Check Collision Between Hero and Enemy

```
if (objects[j]->Tag == "enemy") {
          if (objects[i]->collision.isColliding
     (objects[i]->collision, objects[j]->collision)) {
          objects[i]->makedamage(objects[j]->getattack());
                                        }
```

2.3.2 Hero vs Impassable Terrain

Since the hero's movement is controlled via the keyboard, collision detection occurs every frame, but the movement logic itself is handled in the main function rather than within other classes. As a result, the nested loop in the GameObjectManager class is not suitable for restricting the hero's movement.

To address this, an additional function was implemented to specifically check for collisions between the hero and all impassable terrain objects. This function is called after each movement attempt. If a collision is detected, the movement is canceled, and the hero's position is reverted to its previous state. This approach introduces an additional O(N) computational cost per check, meaning there is still room for optimization in the project's logic.

Listing 13: Function about Check Collision between Hero and Water

```
bool checkwater() {
        for (int i = 0; i < count; i++) {
                if (objects[i]->Tag == "hero") {
                        for (int j = 0; j < count; j++) {
                                if (objects[j]->Tag == "water") {
                                        if (objects[i]->collision.isColliding
                                        (objects[i]->collision,
                                          objects[j]->collision)
                                             &&objects[j]->Active) {
                                             return true;
                                        }
                                }
                        }
                }
        }
        return false;
        //if not colliding,return false so the hero will move
}
```

2.3.3 Hero Projectiles vs NPCs

In the nested loop, if two objects are detected with Tags of "Bullet" and "Enemy" or "Bullet" and "Archr", the NPC takes damage from the bullet. In fact, the bullet's damage value is referenced from a variable stored in the Hero class, which records the hero's attack power. After a collision occurs, the bullet is destroyed. It is first removed from both relevant manager classes, and then deleted from memory to complete the cleanup process.

Listing 14: Check Collision between Hero Projectiles and NPCs

```
if (objects[i]->Tag == "bullet") {
        if (objects[j]->Tag == "enemy" || objects[j]->Tag == "Archr")
    {
    if (objects[i]->collision.isColliding(objects[i]->collision,
    objects[j]->collision)) {
        objects[j]->makedamage(Hero::getInstance().getdamage());
        objects[i]->suicide();
        //remove itself from manager class and then delete ifself
        if (objects[i] == nullptr) {
        //After the bullet is deleted, determine if he is the last obj.
        // If not, skip this loop and continue from the original position.
        break;}
        else {i--;}}}
}
```

2.3.4 NPC Projectiles vs Hero

The logic is similar to that of Hero projectiles vs. NPCs. In the nested loop, when two objects with Tags of "Hero" and "Enemybullet" are detected to be colliding, the NPC projectile is destroyed, and the hero takes a fixed amount of damage as a result of the collision.

2.4 The Hero Attacks

2.4.1 Line Attack

The Hero class keeps track of elapsed time, which is updated every frame. It also maintains another variable to record the shooting cooldown. When the recorded time exceeds the cooldown threshold, a new bullet is generated with an assigned direction and position. In the main function, the hero's state is updated each frame, and the program checks whether a new bullet should be created. If so, a new Bullet instance is generated, and the hero's current position is assigned as the bullet's starting position. The program then calls the enemiesmanager to retrieve all active NPC instances, iterates through them to find the closest enemy, calculates the unit vector from the hero to that enemy, and assigns this vector to the bullet to define its direction.

Listing 15: Hero Shot Bullet Function

```
void Hero::shot(float dt,Hero &hero) {
        shottimecount += dt;

        if (shottimecount >= cooldown) {
                length = 10000;
                cloest = -1;
                shottimecount = 0;
                Bullet* bullet = new Bullet();
                bullet->transform.SetPosition
                (this->transform.GetPositionX() + 40,
                this->transform.GetPositionY() + 60);
                //make sure the new bullet be made in the middle of hero
                for (int i = 0; i < enemiesmanager::getInstance().getcount();
                    i++) {
                        if (length >= sqrt(getDistance
                        (this,
                            enemiesmanager::getInstance().getenemies()[i]))) {

                                cloest = i;
                                length = sqrt(getDistance
                                (this,
                                    enemiesmanager::getInstance().getenemies()[i]));
                        }
                }
                //find the closet enemy
                if (cloest != -1)
                bullet->setdirection(getDirectionX(this,
        enemiesmanager::getInstance().getenemies()[cloest]),
                        getDirectionY(this,
            enemiesmanager::getInstance().getenemies()[cloest]));
                bullet->setmovespeed(hero.getbulletmovespeed());


        }

}
```

2.4.2 A Special Area of Effect (AOE) Attack

Considering that this AOE ability directly targets and operates on NPCs, its implementation can be defined and referenced within the enemiesmanager class. The function only needs to get the maximum number of targets from the Hero instance, then iterate through all NPCs that number of times. In each traversal, it finds the enemy with the highest health, removes it from both the enemiesmanager and GameObjectManager, and then deletes the instance. In the main function, if the 'N' key is pressed, it triggers the enemiesmanager to call the killsomeenemies () function, which searches for and removes the specified number of highest-HP NPCs from the game.

Listing 16: AOE Function

```
 void killsomeenemies(int n) {
    int maxhelath = -1;
    int beenkilled=-1;
    for (int i = 0; i < n; i++) {
        maxhelath = -1;
        for (int j = 0; j < count; j++) {
            if(enemies[j]->gethealth() > maxhelath) {
                maxhelath = enemies[j]->gethealth();
                beenkilled = j;
                        }

        }
        //find the highest-NPCS
        if (beenkilled != -1) {
            enemy* scv = enemies[beenkilled];

            GameObjectManager::getInstance().remove(enemies[beenkilled]);
            remove(enemies[beenkilled]);
            delete scv;
        }
        //if have NPC in the game,kill the highest-NPCs
  }
}
```

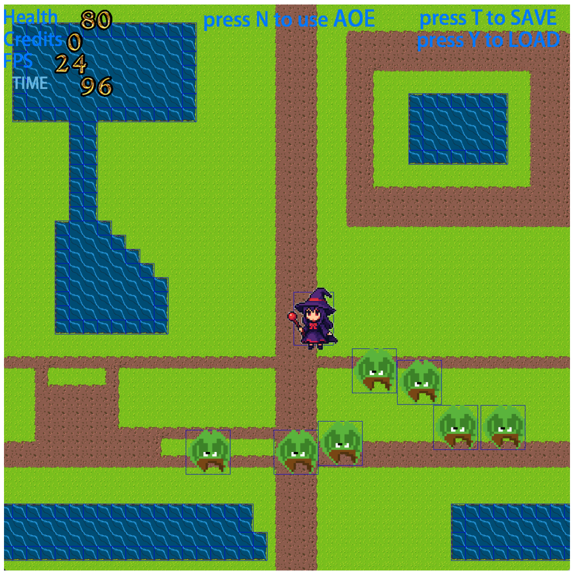


Figure 7: Before Using AOE

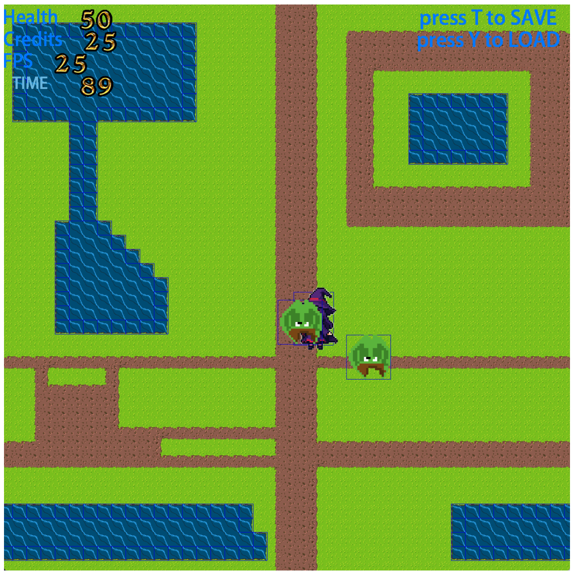


Figure 8: After Using AOE

2.4.3 A Powerup

This project defines two classes, powerup_lineattack and powerup_maxnumber, which inherit from GameObject and are used to increase the bullet movement speed and the maximum number of AOEs, respectively. In the GameObjectManager, during the nested loop for collision detection, if a collision between the hero and these power-ups is detected, the corresponding effect will be triggered. The powerup_lineattack instance reduces the variable related to bullet cooldown in the Hero instance by 10

Listing 17: Detect The Collision between Hero and power_maxnumber

```
if (objects[i]->Tag == "hero") {
    if (objects[j]->Tag == "powerup_maxnumber"
        && objects[i]->collision.isColliding(objects[i]->collision,
    objects[j]->collision)) {
                Hero::getInstance().changemax(1);
                remove(objects[j]);
                if (objects[j] == nullptr) {
                //After the powerup is deleted, determine if he is the last
                    obj.
                // If not, skip this loop and continue from the original
                    position.
                        break;
                          }
                else {
                          j--;
                          }
                }
```

2.5 A Tile-based Method for Displaying the Background

This project uses a 42×42 tile-based map, with each tile measuring 32×32 pixels. The map is arranged sequentially starting from the (0, 0) coordinate in the top-left corner. For an infinite map effect, the system calculates an offset for each row or column of tiles based on the hero's current position, ensuring that the map appears to loop seamlessly as the hero moves. The map's rendering is also influenced by the camera's offset values. During the rendering process, all tiles are iterated through, and their positions are adjusted by adding the appropriate offset values so that they are displayed in the correct sequence. For example, the tile in the first row, third column will have its Y position increased by 64 to ensure proper placement. All tile types are stored in a two-dimensional pointer to Image objects called tiles. Another two-dimensional pointer to int, named mapsave1, stores the tile type index corresponding to each tile position. For instance, if mapsave1[10][10] is 10, it indicates that the tile at the 11th row and 11th column uses the image referenced by tiles[10].

2.5.1 Different Tile Types

The mapsave1 pointer, a two-dimensional array of integers, records the tile type corresponding to each tile on the map. This project provides two selectable maps with different terrain configurations, each containing more than four tile types. Tiles numbered 14–22 represent water areas. To make water act as an obstacle that blocks the hero's movement, the system uses a three-level pointer, watermap, which stores instances of the Water class. When the map is loaded—or when it shifts to simulate an infinite map—each corresponding Water instance has its collision box size set to 32×32 and its collision box position updated accordingly. The Water class inherits from GameObject. Upon initialization, it sets its collision box width and height to 0 and marks its boolean variable Active as false. In total, there are 42×42 Water instances stored in watermap. Only when the value in mapsave1 at the same position is between 14 and 22 does the corresponding Water instance get its collision box set to 32×32 and its Active flag switched to true. Collision detection between Water and the Hero only considers Water instances with Active = true. When the hero attempts to move, the system checks for collisions with these active Water objects. If the movement would cause a collision, it is canceled, and the hero's position reverts to its previous state. In practice, due to the specific map configuration, only tiles numbered 15–22 can actually collide with the hero. Therefore, Water instances corresponding to tile number 14 are not activated and do not have collision boxes assigned. However, since every tile still initializes a Water instance, even non-water tiles are registered in the GameObjectManager. As a result, they contribute to unnecessary performance overhead in each update loop — leaving significant space for optimization in future

Listing 18: Draw Background Function and Update Water Collision Box

```
void draw_entire_background(int** mapsave1,
        GamesEngineeringBase::Window& canvas,
        GamesEngineeringBase::Image* tiles,
        int** offestmapx,
        int** offestmapy, Water*** watermap) {
```

```
        //scvcount++;
        for (int i = 0; i < 42; i++) {
                for (int j = 0; j < 42; j++) {
                        if (mapsave1[i][j] > -1 && mapsave1[i][j] < 24) {
                                if (mapsave1[i][j] >= 15 && mapsave1[i][j] <=
                                    22) {
                                        //update water position and water
                                            collision box position
                                        watermap[i][j]->transform.SetPosition((j
                                            * 32) +
                                                offestmapx[j][i], (i * 32) +
                                                    offestmapy[j][i]);
                                        watermap[i][j]->Update(0,
                                            Camera::GetCamera());

                                }

                                draw_title((j * 32) -
                                    Camera::GetCamera().GetX()
                                        + offestmapx[j][i], (i * 32) -
                                            Camera::GetCamera().GetY()
                                        + offestmapy[j][i], canvas,
                                            tiles[mapsave1[i][j]]);
                        }
                }
        }
}
```

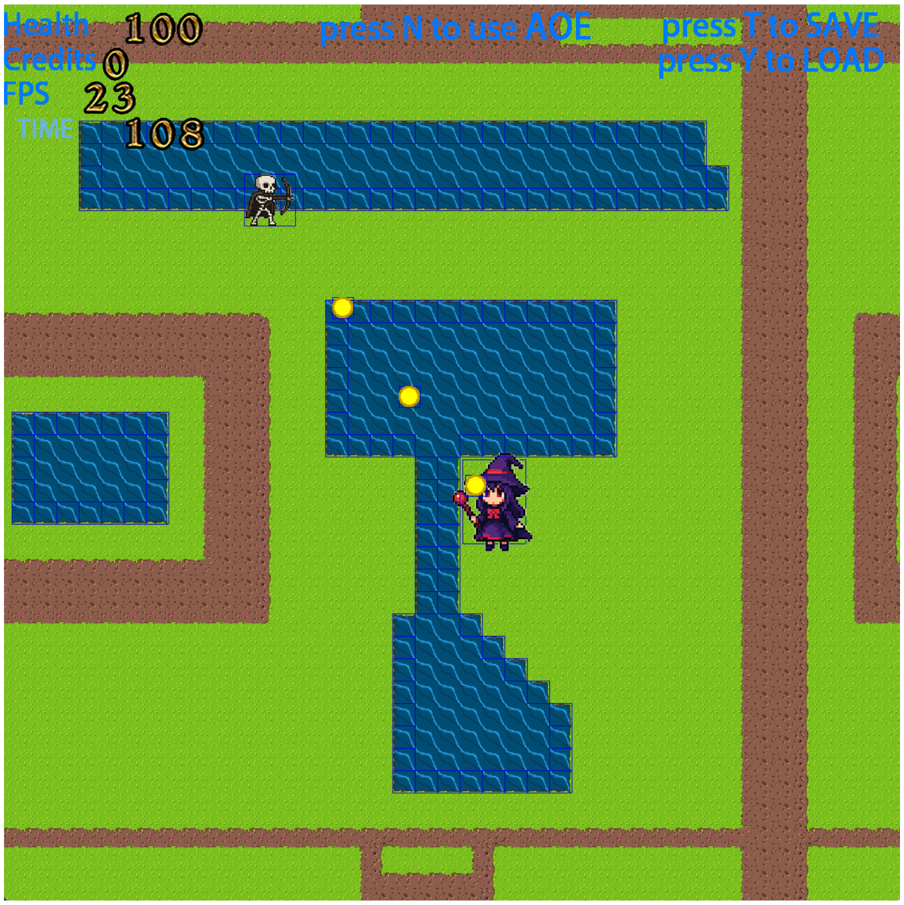


Figure 9: Impassable Terrain

2.5.2 Data Driven Level Loading

The map file follows the format provided in the course's Resources folder. It contains a 42×42 grid of integer data, representing the tile types across the map. During the loading process, the program reads these integer values from the file and stores them into the two-dimensional integer pointer mapsave1. When the map is drawn, the program iterates through all positions in mapsave1 and retrieves the corresponding tile image from the two-dimensional image pointer tiles, rendering each tile based on its mapped value and position. All tile images are stored in tiles, which holds pointers to the image resources used for rendering the map. The project includes two different maps, each with distinct terrain settings. Depending on the player's selection, the program loads a different map file and stores its data into mapsave1. Below is the implementation for one of the two maps. The other map follows the same principle and structure, differing only in the input file and tile data used for initialization.

Listing 19: Read Map.txt Function

```
if (loadpagechose == 1 || loadpagechose == 3) {
        ifstream file("../Resources/tiles.txt");

        string line;
        for (int linenumber = 0; getline(file, line);) {
                if (linenumber >= 6) {
                        //lins between 0-5 is other is the introduce of map
                        int columnline = 0;
                        int stack = -1;
                        for (int i = 0; i < line.size(); i++) {
                                if (line[i] == ',') {
                                        mapsave1[linenumber - 6][columnline]
                                            = stack;
                                        columnline++;
                                        stack = -1;
                                }
                                else {
                                        //calculate the number from string
                                        if (stack == -1) {
                                                stack = line[i] - '0';
                                        }
                                        else {
                                                stack = stack * 10 + (line[i]
                                                    - '0');
                                        }
                                }

                        }
                }
                linenumber++;

        }
        file.close();
        for (int i = 0; i < 42; i++) {
                for (int j = 0; j < 42; j++) {

                        if (mapsave1[i][j] >= 15 && mapsave1[i][j] <= 22) {
                                watermap[i][j]->collision.SetCollision(0, 0,
                                    32, 32);
                                //active water
                        }
                        else {
                                watermap[i][j]->collision.SetCollision(0, 0,
                                    0, 0);
                                watermap[i][j]->Active = false;
                        }
                }
        }
}
```

2.5.3 A Version of the World which is Infinite

There are two important components in the implementation of the infinite map. One is the virtual camera offset, which has been discussed in detail in the Camera section above. The other is continuously updating the actual position of each map tile based on the hero's position. The update principle is very simple. For example, when the hero moves to the right, the leftmost column of tiles is moved to the right by one map width, which is 42×32pixels, to complete the position update. Two two-dimensional integer pointers, mapmapoffsetx and mapmapoffsety, record the offset of each tile in the x and y directions, respectively. The specific offset situations are divided into four cases: positive x-direction, negative x-direction, positive y-direction, and negative y-direction. Below, the positive x-direction is explained in detail. First, the difference between the hero's current position and the center of the map needs to be calculated. Note that this is not the window center, but half the map length, which is 42×32. When the hero is at the map center, no offset is needed. If the hero is not at this position, the offset number and the number of affected tiles must be calculated. Here, the remainder and quotient of the hero's movement divided by the map length are computed separately. The quotient represents how many columns from left to right need to be shifted by one map width, while the remainder indicates the amount by which all tiles need to be offset to the right. Using this method, the offset required for all tiles can be calculated to ensure that the map loops continuously within the camera view.

Listing 20: Update Map Position Function

```
void changemao(Hero& hero, int** mapmapoffestx, int** mapmapoffesty)
    {//change map offest
        int countx = ((int)hero.transform.GetPositionX() - 672) / (42 * 32);
        int county = ((int)hero.transform.GetPositionY() - 672) / (42 * 32);
        int offestx = ((int)hero.transform.GetPositionX() - 672) % (42 * 32)
            / 32;
        int offesty = ((int)hero.transform.GetPositionY() - 672) % (42 * 32)
            / 32;

        for (int i = 0; i < 42; i++) {
                for (int j = 0; j < 42; j++) {
                        //clean its
                        mapmapoffestx[i][j] = 0;
                        mapmapoffesty[i][j] = 0;
                }
        }
        if (offestx < 0 || countx < 0) {
                //hero in the left
                for (int i = 41; i >= 0; i--) {
                        for (int j = 41; j >= 0; j--) {
                                if (j > 41 + offestx) {
                                        mapmapoffestx[j][i] = (-1 + countx *
                                            1) * 42 * 32;
                                }
                                else {
                                        mapmapoffestx[j][i] = (countx * 1) *
                                            42 * 32;
                                }
                        }
                }
        }
        if (offestx >= 0 || countx > 0) {
                //hero in the right
                for (int i = 0; i < 42; i++) {
                        for (int j = 0; j < 42; j++) {

                                if (j < offestx)
```

```
                                        mapmapoffestx[j][i] = (1 + countx *
                                            1) * 42 * 32;
                                else {
                                        mapmapoffestx[j][i] = (countx * 1) *
                                            42 * 32;
                                }
                        }
                }
        }
        if (offesty >= 0 || county > 0) {
                //hero in the top
                for (int i = 0; i < 42; i++) {
                        for (int j = 0; j < 42; j++) {
                                if (j < offesty) {
                                        mapmapoffesty[i][j] = (1 + county *
                                            1) * 42 * 32;
                                }
                                else {
                                        mapmapoffesty[i][j] = (county * 1) *
                                            42 * 32;
                                }
                        }
                }
        }
        if (offesty < 0 || county < 0) {
                //hero in the down
                for (int i = 0; i < 42; i++) {
                        for (int j = 41; j >= 0; j--) {
                                if (j > 41 + offesty) {
                                        mapmapoffesty[i][j] = (-1 + county *
                                            1) * 42 * 32;
                                }
                                else {
                                        mapmapoffesty[i][j] = (county * 1) *
                                            42 * 32;
                                }

                        }
                }
        }

}
```

2.5.4 A Version with a Fixed Boundary

The finite map is implemented based on the infinite map system, but its implementation is simpler. First, the tile position updates from the infinite map are removed, so map tiles no longer have any offsets. Next, by restricting the hero's position and the camera's offset values, the map is constrained to a finite area. It is important to note that because the map is fixed, the world beyond the map boundaries is empty. Therefore, when the hero approaches the edge, the camera stops following the hero and only displays up to the map boundary. As a result, the camera offset limits must be set slightly smaller than the hero's movement limits to prevent displaying empty space beyond the map.

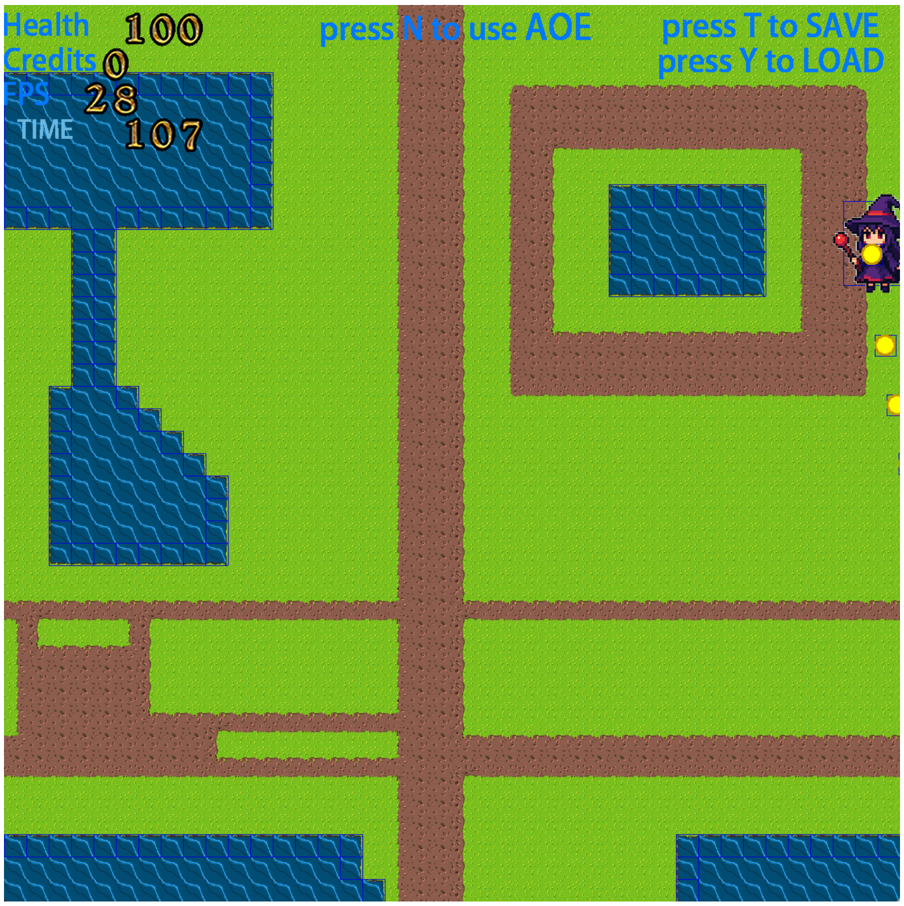


Figure 10: Hero with Fixed Map

2.6 Game Level

Here is a brief introduction to the scene switching in the project. Visually, the project contains six scenes: the start screen, end screen, gameplay screen, map selection screen, save screen, and load screen. Outside the main game loop, there is a loop that controls scene transitions. When entering the gameplay scene, a timer starts, and the game ends after 120 seconds. If the timer expires or the hero dies, the scene switches to the end screen, and a boolean value is passed to indicate whether the player won or lost. It is important to note that the timer only starts when entering the gameplay scene and is reset each time the scene is entered. Although the project also contains a separate thread running a continuous timer to prevent accidental keyboard input, this timer is unrelated to the gameplay timing.

2.6.1 Chosen Map

In the map selection screen, four options are presented: Infinite Terrain 1, Infinite Terrain 2, Fixed Terrain 1, and Fixed Terrain 2. An integer variable is used to record which map the player selects. In the gameplay scene, the program first loads the selected map. During the game loop, it uses the value of the selection variable to determine whether to apply the fixed map rules—restricting the camera and hero position updates—or the infinite map rules, allowing unrestricted moving.

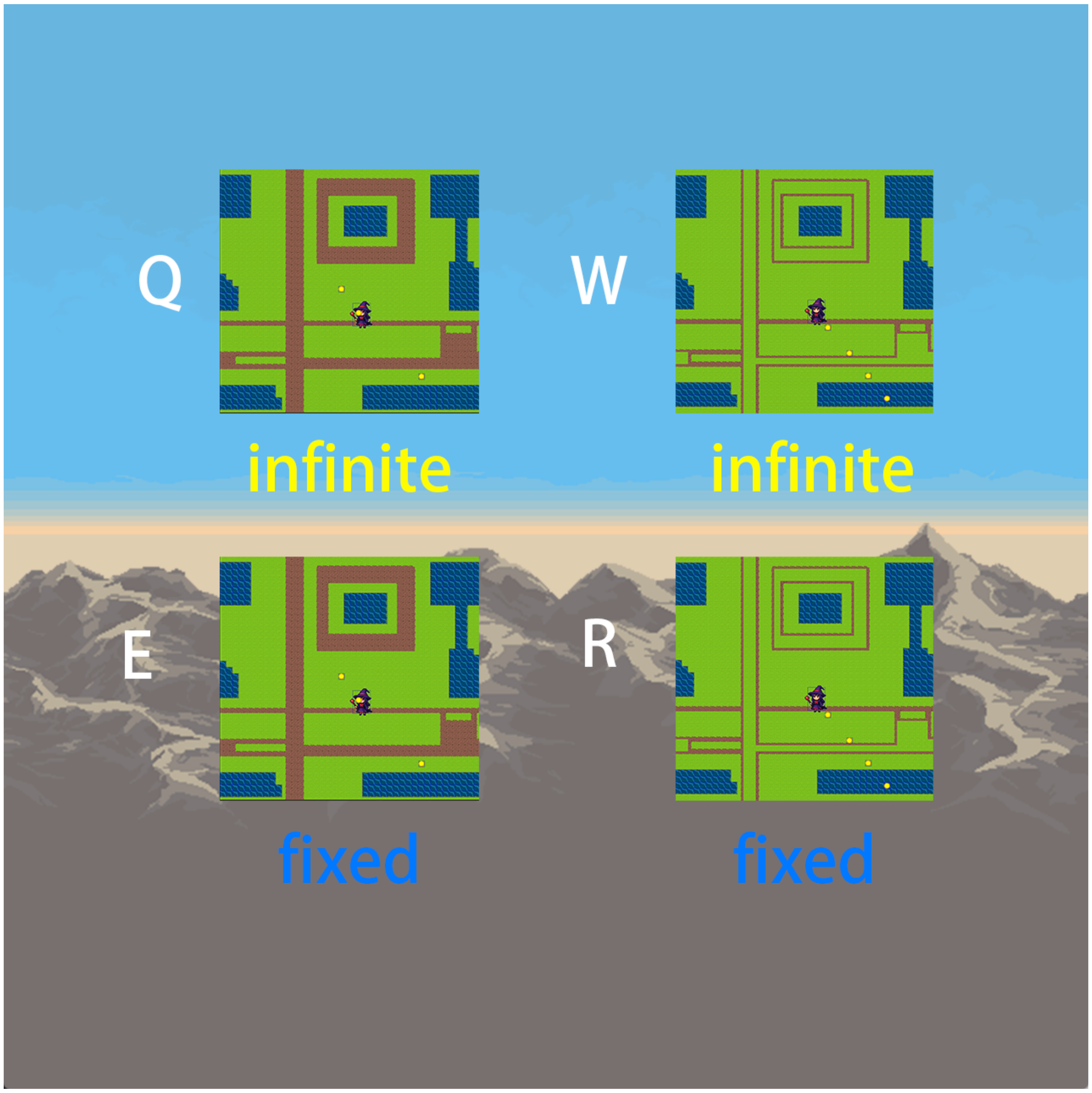


Figure 11: Chosen Map Page

2.6.2 Score and FPS Shown

The score and FPS are displayed during gameplay. When an enemy unit dies, the score automatically increases by 5 points. The FPS is refreshed once per second, based on the number of game loop iterations in the previous second. All text in the game is rendered using images. Since numbers need to be displayed dynamically, a single image containing digits 0–9 is used. Depending on the number of digits and their size, different portions of this image are displayed at different positions on the screen to represent the current values.

Listing 21: Draw Number Function

```
void draw_numbers(float x, float y, GamesEngineeringBase::Window& canvas,
        GamesEngineeringBase::Image& image, int number) {
        int numbernow;
        int count = 0;
        int numbers[10];
        if (number == 0) {
                //draw 0 when the number is 0
                for (int i = 0; i < 39; i++) {
```

```
                for (int j = 0; j < 50; j++) {
                        if (image.alphaAt(i , j)) {
                                canvas.draw(i + x, j + y, image.at(i
                                    , j));
                        }
                }
        }
}
while (number >0) {

        numbers[count] = number % 10;
        //save the number in different digits
        count++;
        //count the The number of digits
        number = number / 10;

}
for (int u = count-1; u >= 0;u-- ) {
        for (int i = 0; i < 39; i++) {
                for (int j = 0; j < 50; j++) {
                        if (image.alphaAt(i + (numbers[u] * 39), j)) {
                                canvas.draw(i + (count - u) * 39 + x,
                                        //draw the number based on
                                            digits
                                j + y, image.at(i + (numbers[u] *
                                    39), j));
                        }
                }
        }
}


}
```

2.7 Save and Load

The game is data-driven, meaning that as long as all key game data are recorded, saving and loading can be achieved by restarting the game and reloading the previous data. All units in the game inherit from GameObject, so saving their state only requires defining a virtual function in GameObject, and then implementing the specific functionality in each subclass to handle saving for different instances. Saving is performed by iterating through all stored objects in the GameObjectManager and executing their save functions. Besides units, several other key data need to be saved, including the camera offset values, the selected scene, the game score, the data about enemiesmanager class,and the remaining game time. When loading a save, all existing GameObject instances are first cleared using the GameObjectManager, except for the Hero instance, which is not deleted. The EnemiesManager and Bulletmmanager also reset their counters. Data saving and loading are handled using JSON. During loading, the program determines the object type using its Tag and creates a new instance. GameObject has a virtual load() function, which each subclass calls after instantiation to assign the saved data to the object. The loaded data for the selected scene, score, and remaining time are directly assigned to the relevant variables in the main function, while the camera offset is assigned to the Camera singleton instance and the data about enemiesmanager is assigned to the enemiesmanager singleton instance, which is not deleted before loading. The Hero instance works similarly; as a GameObject subclass, it also uses load() to restore its data. After loading, the map is reloaded based on the saved landscape number, and the previously mentioned mapsave1 and watermap arrays in the main function are cleared and rebuilt. The project provides three save slots, which can be used during gameplay or accessed directly from the start screen and end screen. In the gameplay scene, pressing T saves the game, while pressing Y loads a save. When loading from the start screen or end screen, the game first transitions to the gameplay scene, completes initialization, then goes to the load scene. After selecting a save file, all instances are cleared and reloaded accordingly.

Listing 22: Hero Save Function

```
void save(json& obj) override {
        obj.push_back({
                {"Tag","hero"},
                {"position_x",transform.GetPositionX()},
                {"position_y",transform.GetPositionY()},
                {"health",health  },
                {"movespeed",movespeed },
                {"pojectilespeed",pojectilespeed },
                {"abilitytimecount",abilitytimecount },
                {"abilitymax",abilitymax },
                {"abilitycooldown",abilitycooldown },
                {"bulletmovespeed",bulletmovespeed},
                {"cooldown",cooldown},
                {"shottimecount",shottimecount},
                {"invincible",invincible},
                {"invincibletimecount",invincibletimecount}

                });
}
```

Listing 23: Hero Load Function

```
void load(json& obj) override {
        transform.SetPosition(obj["position_x"], obj["position_y"]);
        health = obj["health"];
        collision.SetCollision(transform.GetPositionX(),
            transform.GetPositionY());
        movespeed = obj["movespeed"];
        pojectilespeed = obj["pojectilespeed"];
        abilitytimecount = obj["abilitytimecount"];
        abilitymax = obj["abilitymax"];
        abilitycooldown = obj["abilitycooldown"];
        bulletmovespeed = obj["bulletmovespeed"];
        cooldown = obj["cooldown"];
        shottimecount = obj["shottimecount"];
        invincible = obj["invincible"];
        invincibletimecount = obj["invincibletimecount"];
}
```

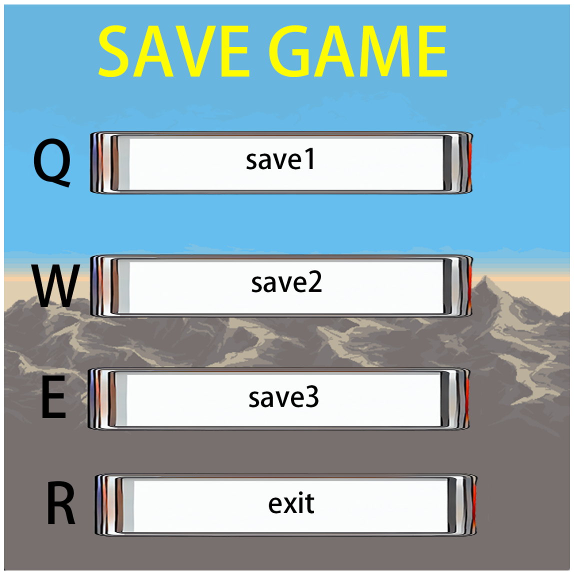


Figure 12: Save Page

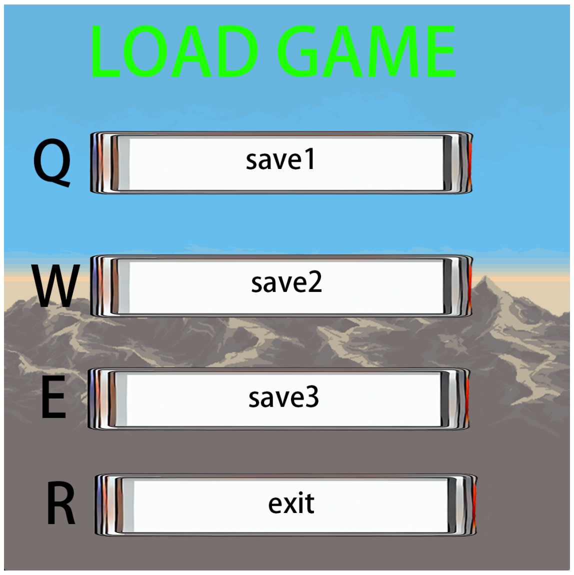


Figure 13: Load Page

3 Analysis of Performance

This project uses a 1280×1280 window, which means that at least 1,638,400 pixels need to be drawn on the background every frame. When the game first starts running, the frame rate is approximately 28 FPS. However, if the background is not drawn, the frame rate at the start of the game can reach around 88 FPS. For the sake of clearer performance data, most of the following tests are conducted without rendering the background. - When only the hero can move up, down, left, or right, the frame rate stabilizes at around 101 FPS.

After introducing bullets, the frame rate drops to approximately 98 FPS. Since bullets are time-limited, their number does not accumulate indefinitely, so in this project, they do not cause significant performance overhead.

After introducing enemies, when there are 10 enemies, the frame rate drops to 72 FPS. At this point, the main performance cost comes from the EnemiesManager, which must calculate the vector from each enemy to the hero and update both the enemy positions and their collision boxes. When there are 20 enemies, the frame rate drops to 58 FPS. With 30 enemies, the frame rate drops to 49 FPS. During testing, it was observed that as units gradually enter the screen, the frame rate drops by 10–15 FPS compared to when they are off-screen, indicating that rendering is indeed very performance-intensive. Next, collision detection was enabled for testing. Once collisions are activated, the additional performance overhead comes from several sources. In EnemiesManager, there is an $O(N^2)$ cost for checking collisions between each pair of enemies. In GameObjectManager, there is an $O(N^2)$ cost for checking collisions among all GameObjects.

Test Method	Enemy Number	FPS
No Collision Detection	10	72
No Collision Detection	20	58
No Collision Detection	30	49
Collision Detection Enabled	10	72
Collision Detection Enabled	20	59
Collision Detection Enabled	30	52
Without Rendering, No Collision Detection	10	104
Without Rendering, No Collision Detection	20	97
Without Rendering, No Collision Detection	30	94
Without Rendering, With Collision Detection	10	102
Without Rendering, With Collision Detection	20	95
Without Rendering, With Collision Detection	30	92
Without Rendering, With Collision Detection and Water Tiles	10	95
Without Rendering, With Collision Detection and Water Tiles	20	88
Without Rendering, With Collision Detection and Water Tiles	30	83
Normal Rendering (Including Background)	10	25
Normal Rendering (Including Background)	20	22
Normal Rendering (Including Background)	30	21

Table 1: Performance Test Results Under Different Conditions

Undoubtedly, rendering consumes far more performance than logic calculations. When rendering NPCs while performing collision detection, the frame rate does not show a significant difference despite the $O(N^2)$ collision checks. This is mainly because collision detection sometimes blocks NPCs from entering the screen, preventing them from being drawn and actually saving performance. Additionally, with collision detection enabled, bullets are immediately destroyed, reducing the number of active units on the field. These factors together lead to minimal observable changes in FPS. In tests without rendering, enabling collision detection slightly lowers the FPS compared to disabling it. This is because the GameObjectManager skips loops for objects that do not require collision checks, making the overhead closer to $O(N)$. Since the number of NPCs is small and the collision logic is not complex, the additional performance cost is minimal. However, when water tiles are introduced, the performance drop becomes significant. This is due to a poor design choice: the system creates 42×42 Water objects but only activates those corresponding to actual water tiles. While most of these instances do not participate in calculations, each still requires a boolean check, resulting in O(N) overhead, which greatly slows down the game. In the final fully operational game, the number of enemies does affect the frame rate, but because rendering consumes most of the resources, the FPS remains very low overall.

4 Limitations

NPCs can get stuck during movement due to collisions. Although the assignment did not require collisions between NPCs, I implemented it anyway. From discussions with classmates, I learned that using circular collision shapes instead of the square ones I used can effectively prevent this issue. However, it's inconvenient for me to make that change, and a square collision box theoretically shouldn't cause this problem. The issue could be solved by introducing a rigid body system to simulate realistic physical collisions, or by implementing a pathfinding algorithm like A*. In my game, movement and collisions are calculated using simple methods with no physics simulation, which leads to the NPCs getting stuck. Unfortunately, at my current skill level, it's very difficult to implement proper physics. there are performance issues with water collisions. I used a triple pointer to Water objects, storing 42×42 water tiles. I realize that I could have activated only the water tiles present on the map and deleted the inactive ones so they wouldn't consume resources during collision checks. However, doing so would require changes to the save/load system, and by the time I considered it, modifying the code was too complex. When I first added water collisions, I tried deleting unnecessary Water objects, but this caused null pointer issues. Since the program still ran correctly, I decided to postpone this optimization, and it was never addressed in the final version.

5 Revised Development Strategy

Firstly, the project structure could be improved. For example, the Bullet and Enemybullet classes represent the hero's bullets and NPC bullets, respectively, and Enemybullet is the subclass of Bullet. A more reasonable design would be to have a single Bullet base class with two subclasses, or even use a single class and differentiate by Tag, since the differences are minimal. The three manager classes are somewhat coupled, a more efficient approach might be to use one central manager loop to update all instances. Collision detection should adopt more efficient methods, such as quadtrees, because the $O(N^2)$ approach used in this project causes significant performance overhead. In this project, water collisions also caused serious performance issues. A better approach would be to remove all unused Water instances, or find a more suitable method altogether. Smart pointers should be used for objects—for example, the optimization for Water instances was abandoned due to null pointer issues. Some parts of the code lack abstraction, which reduces readability. Finally, rendering causes significant performance overhead. Without GPU acceleration, one possible optimization is to avoid redrawing areas that have not changed, which could significantly reduce the rendering overhead.

6 Conclusion

In summary, this project successfully implemented a Vampire Survivors–style game using C++ under the GamesEngineeringBase framework. The project fulfilled key requirements such as a virtual camera, collision detection, infinite maps, NPC generation and control, and a JSON-based save/load system. Basic optimizations were carried out to ensure that the game runs smoothly for a 120-second session. Additionally, by analyzing the game's performance, potential space for further optimization were identified. During the development process, I also gained a deeper understanding of the game framework and practical usage of C++ features. Overall, the project successfully achieved its intended design objectives.

Project Repository

The source code for this project is available on GitHub: `https://github.com/AODINGLVP/assignment.git`