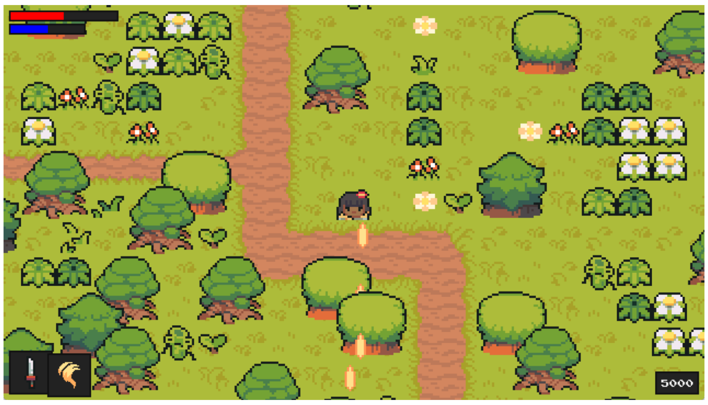
Collision Detection in the Zelda

01 Introduction

1.1 Intro to the Game Project

Just like the famous 2D top-down game *The Legend of Zelda: A Link to the Past*, the game project chosen for this report is a 2D classic Zelda-style game, as shown in the screenshot below:



Players can use the arrow keys to move the character in eight directions, press the spacebar to attack enemies, and use the Ctrl key to unleash magical attacks on opponents. Meanwhile, you can switch melee weapons using the Q key, and switch the type of magic to either fire magic or healing magic using the E key. Within this context, the interactions between the princess and obstacles such as grass and trees, as well as between the princess and monsters, are the primary focus of this report, which explores the topic of **collision detection**.

1.2 The Chosen Algorithm and Its Relevance to the Course

The algorithm chosen for analysis in this report is the **collision detection algorithm**. It is not a specific, pre-implemented algorithm but rather a method designed to solve a class of problems: *We know how to draw a sprite and blit the `sprite.image` into a rectangle on the display surface. But how should sprites interact with each other? What if one sprite "walks" into another? How do we know if they "bounce" into each other?*

In other words, collision detection is a form of **physical simulation** that determines whether and how objects interact, based solely on mathematical representations of those objects. For this reason, the **collision detection algorithm** is closely tied to the topics covered in our course.

---

02 Algorithm Explanation

2.1 Theory

From a macro perspective, the **collision detection algorithm** is a 4-step plan that includes the following steps:

1. Reduce the number of checks that need to be made
2. Quickly test sprites: discard those that can't possibly collide
3. Thoroughly test remaining sprites to detect collisions
4. Resolve collisions: Do whatever -- explosions, bounces, etc.

For each step, there are many implementation details worth noting.

2.1.1 Step 1: Reduce the Number of Checks That Need to Be Made

The optimization focus of collision detection is to improve efficiency by reducing unnecessary calculations while ensuring algorithm accuracy and normal game functionality.

First, based on game-specific rules: stationary objects do not need to detect collisions between two objects that won't move; or the game has specific interaction rules where certain objects can pass through each other, such as "friendly fire" between teammates.

Secondly, distance-based optimization: objects geographically far apart do not need collision detection; or for off-screen objects, collision detection can be ignored (if not handled properly, anomalies may occur when the player's perspective switches back).

Furthermore, rules based on specific areas: some objects only exist in specific areas, therefore no collision detection is needed between them and objects in other areas.

For object partitioning, there are some classic algorithms:

- Spatial partitioning, such as Quadtree, which divides the world into four regions and recursively subdivides; there are also other algorithms like Octree, BSP, etc.
- Game-based partitioning, designing specific partitioning schemes according to game mechanics. For example, in racing games, the track can be divided into multiple sections, tracking the section each car is in.
- Automated partitioning, using algorithms to automatically partition and further optimize efficiency; including Goldsmith-Salmon and Bottom-Up N-ary Clustering methods.

2.1.2 Step2: Quickly Test Sprites

The core of quick testing lies in utilizing simple, efficient detection methods to determine whether potential collisions might occur, rather than directly calculating whether a collision has actually happened. This involves two basic principles: judging whether a collision is possible, quickly excluding most situations that will not collide, instead of directly confirming whether a collision truly occurs; and selecting appropriate quick testing methods based on the shape and representation of objects in the game.

Here are some classic quick testing methods:

- Circle / Sphere Test: Wrapping objects in a circular (2D) or spherical (3D) shape, where many objects are naturally suitable for circular representation (like asteroids or Mario's turtle shell); if two circles intersect, their center distance must be less than the sum of their radii

- Bounding Box Test: Including AABB (Axis-Aligned Bounding Box, using rectangles aligned with coordinate axes as object boundaries), OOBB (Object-Oriented Bounding Box, using rectangles consistent with object orientation); or utilizing the Separating Axis Theorem (SAT), where if an axis exists that prevents two objects from overlapping, they definitely do not collide
- Capsule Test: When simulating humanoid characters, using a rectangle with two semi-circles (capsule shape), then performing bounding box tests on the rectangular part and circular tests on the two semi-circular parts

Of course, quick testing can be combined with geographic optimizations (such as off-screen object detection or partitioning) to further improve performance, which will not be elaborated here.

2.1.3 Step3: Thoroughly Test Remaining Sprites to Detect Collisions

After quick testing, to further determine whether two objects truly collide and precisely identify the time and location of collision, we primarily introduce the Swept Circle/Sphere Algorithm.

Define the parametric positions of two objects:

$$P(t) = P_0 + v_p t$$

$$Q(t) = Q_0 + v_q t$$

Where $t \in [0, 1]$ represents the time proportion between two frames.

The condition for collision between two circles/spheres is that their distance equals the sum of their radii:

$$|| \, P(t) - Q(t) \, ||= rp + rq$$

By substituting the parametric formula, expand the distance equation into a quadratic equation:

$$(A \cdot A) + 2(A \cdot B)t + (B \cdot B)t^2 - (r_p + r_q)^2 = 0$$

Where: $A = P_0 - Q_0$, $B = v_p - v_q$, $r_p + r_q$ is the sum of radii of two circles/spheres.

Thus, we can obtain the coefficients of the quadratic equation:

- $a = B \cdot B$
- $b = 2(A \cdot B)$
- $c = A \cdot A - (r_p + r_q)^2$

Use the discriminant Δ to judge the existence of solutions, and when $\Delta \geq 0$, use the quadratic equation solution formula to calculate t, and check whether t satisfies $t \in [0, 1]$.

2.1.4 Step4: Resolve Collisions

That is, responding to a collision.

Simple response requires only knowing that a collision has occurred: for example, when a bullet collides with an enemy ship, make the ship explode, or when a bullet collides with a player, reduce the player's health points; it can also generate complex responses, such as elastic collisions

(Elastic Collisions), where the object's velocity can be adjusted through simple reflection formulas.

I'll translate the entire document section from Chinese to English, maintaining the original formatting:

2.2 Implementation

2.2.1 Collision Detection for Obstacles

As mentioned in the previous theoretical discussion, in this game design, collision detection for obstacles was partially optimized: we only focus on classes that might collide with obstacles, which in this implementation are the Player and Enemy classes, both inheriting from the Entity class. This class has two key functions for obstacle collision detection:

```python
def move(self,speed):
    if self.direction.magnitude() != 0:
        self.direction = self.direction.normalize()

    self.hitbox.x += self.direction.x * speed
    self.collision('horizontal')
    self.hitbox.y += self.direction.y * speed
    self.collision('vertical')
    self.rect.center = self.hitbox.center

def collision(self,direction):
    if direction == 'horizontal':
        for sprite in self.obstacle_sprites:
            if sprite.hitbox.colliderect(self.hitbox):
                if self.direction.x > 0: # moving right
                    self.hitbox.right = sprite.hitbox.left
                if self.direction.x < 0: # moving left
                    self.hitbox.left = sprite.hitbox.right

    if direction == 'vertical':
        for sprite in self.obstacle_sprites:
            if sprite.hitbox.colliderect(self.hitbox):
                if self.direction.y > 0: # moving down
                    self.hitbox.bottom = sprite.hitbox.top
                if self.direction.y < 0: # moving up
                    self.hitbox.top = sprite.hitbox.bottom
```

The collision detection logic here is: the player and enemy movements have increments in horizontal and vertical directions. If there's an increment in either direction, it checks the collision situation with obstacles in that direction. If a collision occurs, it responds accordingly.

Of course, in this game design, all obstacles are simply grouped into the obstacle_sprites class, without further subdivision. For example, in this game, there are lush trees on grasslands, while only dead trees exist in the desert. It might be possible to implement a system where if the player is on grassland, only collisions with lush trees are checked, while in the desert, only collisions with dead trees are considered.

It's worth noting that this design uses a hitbox: for players and enemies, the actual hitbox when checking collisions with obstacles is slightly smaller than the loaded image (64 × 64), which creates an occlusion effect that is more visually realistic. The occlusion effect will be demonstrated in the third section, "Execution and Observations".

2.2.2 Collision Detection for Player and Enemy

The main code for collision detection between player and enemy is as follows:

```
self.visible_sprites.update()
self.visible_sprites.enemy_update(self.player)
self.player_attack_logic()
```

The first update will update the state of all visible sprites (including player and enemy) based on keyboard input.

The second enemy_update receives the updated self.player information and judges all sprites belonging to the enemy class. The update logic for each enemy-class sprite is determined by the following code:

```
# this enemy_update method belongs to class Enemy
def enemy_update(self,player):
    self.get_status(player)
    self.actions(player)
def get_status(self, player):
    distance = self.get_player_distance_direction(player)[0]

    if distance <= self.attack_radius and self.can_attack:
       if self.status != 'attack':
          self.frame_index = 0
       self.status = 'attack'
    elif distance <= self.notice_radius:
       self.status = 'move'
    else:
       self.status = 'idle'
def actions(self,player):
    if self.status == 'attack':
       self.attack_time = pygame.time.get_ticks()
       self.damage_player(self.attack_damage,self.attack_type)
       self.attack_sound.play()
    elif self.status == 'move':
       self.direction = self.get_player_distance_direction(player)[1]
    else:
       self.direction = pygame.math.Vector2()
def damage_player(self,amount,attack_type):
    if self.player.vulnerable:
       self.player.health -= amount
       self.player.vulnerable = False
       self.player.hurt_time = pygame.time.get_ticks()
       self.animation_player.create_particles(attack_type,self.player.rect.center,
[self.visible_sprites])
```

First, it retrieves the current distance information between the enemy and player (distance, direction), which is determined based on rect.center.

Then, the enemy judges based on this distance information. If the distance is within the enemy's attack radius (attack_radius, a predefined constant in settings.py), it sets the enemy's current status to 'attack'. If outside the attack radius but within the enemy's detection radius, it sets the status to 'move'; otherwise, the enemy takes no action.

The code shows variables like can_attack and frame_index, which actually control Animation. However, since Animation is not directly related to this report, we won't elaborate further, only showing the main collision detection logic.

Finally, it performs actions based on the current enemy status. If 'attack', it records attack_time and calls damage_player to attack the player (the game even implements sound effects!). If 'move', it updates the enemy's direction to point towards the player; otherwise, it initializes the direction to Vector2(), meaning no movement.

The third self.player_attack_logic() implements the player's attack on enemies:

```python
def player_attack_logic(self):
    if self.attack_sprites:
        for attack_sprite in self.attack_sprites:
            collision_sprites = 
pygame.sprite.spritecollide(attack_sprite,self.attackable_sprites,False)
            if collision_sprites:
                for target_sprite in collision_sprites:
                    if target_sprite.sprite_type == 'grass':
                        pos = target_sprite.rect.center
                        offset = pygame.math.Vector2(0,75)
                        for leaf in range(randint(3,6)):
                            self.animation_player.create_grass_particles(pos - offset,
[self.visible_sprites])
                        target_sprite.kill()
                    else:
                        target_sprite.get_damage(self.player,attack_sprite.sprite_type)
def get_damage(self,player,attack_type):
    if self.vulnerable:
        self.hit_sound.play()
        self.direction = self.get_player_distance_direction(player)[1]
        if attack_type == 'weapon':
            self.health -= player.get_full_weapon_damage()
        else:
            self.health -= player.get_full_magic_damage()
        self.hit_time = pygame.time.get_ticks()
        self.vulnerable = False
```

It checks for collisions between the player's current weapon attack_sprites (one of five melee weapons or fire magic attack) and all attackable_sprites, filtering out collision_sprites.

Then, for each sprite in collision_sprites, it handles the corresponding response: if it's 'grass', it simply kills the grass sprite.
Here, it even implements an animation effect when killing grass, with leaves scattering!

If it's another type (which in this game is actually just enemy, as the attackable group only contains grass and enemy), it calls get_damage to update the enemy's information.

The game also implements a knockback effect when an enemy is hit, in the initial update section:

```python
def update(self):
    self.hit_reaction()
    self.move(self.speed)
    self.animate()
    self.cooldowns()
    self.check_death()

def hit_reaction(self):
    if not self.vulnerable:
       self.direction *= -self.resistance
if not self.vulnerable:
    if current_time - self.hurt_time >= self.invulnerability_duration:
       self.vulnerable = True
def cooldowns(self):
    current_time = pygame.time.get_ticks()
        # ......
    if not self.vulnerable:
       if current_time - self.hurt_time >= self.invulnerability_duration:
          self.vulnerable = True
```

The knockback distance parameter resistance is also defined in settings.py.

Meanwhile, there is also a function called cooldowns, which implements an "invincibility" effect: each time the enemy or player is attacked, they have an invincibility duration during which they cannot be attacked again.

Thus, the implementation of collision detection in this game design is fully analyzed! The specific performance of collisions in the game will be demonstrated in the third section "Execution and Observations"!

2.3 Importance in Game Development

Collision detection is fundamental to implementing core game mechanics, because we must handle interactions between game entities (players, enemies, items), and at the same time, we are not just detecting collisions, but more importantly, performing collision detection efficiently, reducing unnecessary computational checks and overhead.

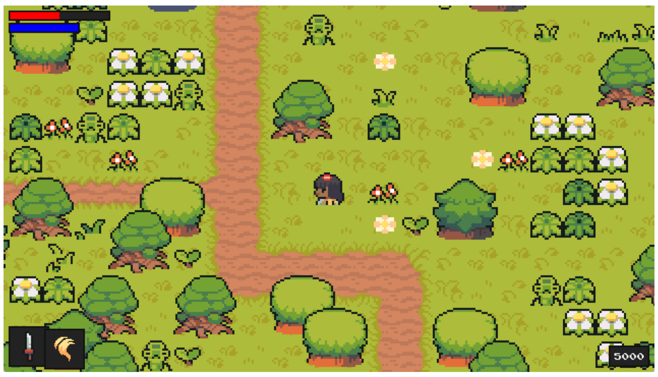
Simultaneously, optimizing the game experience is an absolutely indispensable aspect, such as visual realism, where characters are partially obscured by trees, or monsters are knocked back when attacked.

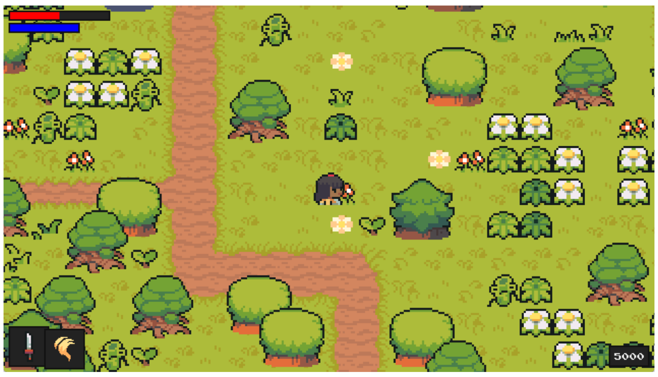
Only under such a collision detection algorithm can we truly implement effective collision detection and create a rich gaming experience for players.

---

03 Execution and Observations

For setting up the code, you only need a Python environment and Pygame installed to run it! Next, we will demonstrate various collisions:





Figures 4-1 4-2: For objects that are visible but not obstacles, you can directly "walk over" them

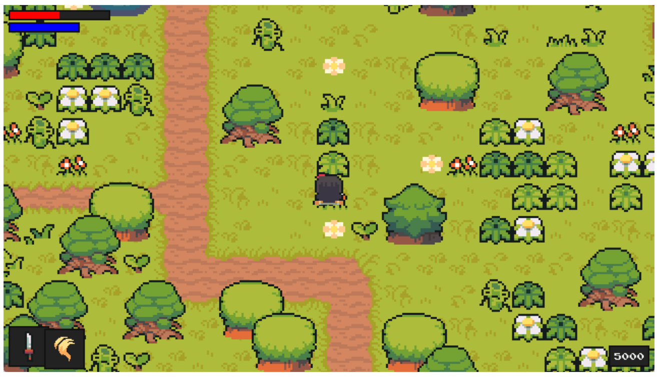


Figure 4-3: For objects that are visible and are obstacles, you will be blocked and cannot pass

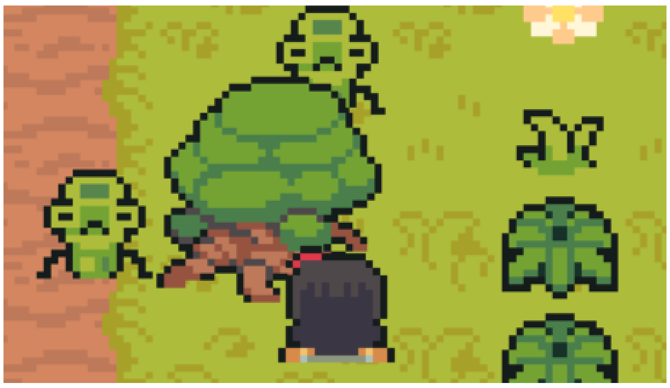


Figure 4-4: When the princess is below the tree, she can partially obscure the tree's vision

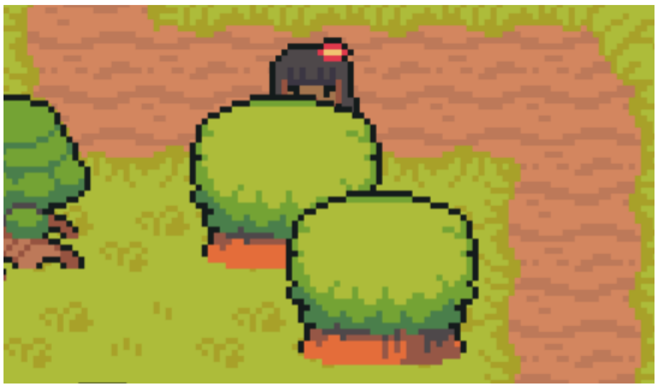


Figure 4-5: When the princess is above the tree, she is partially obscured by the tree

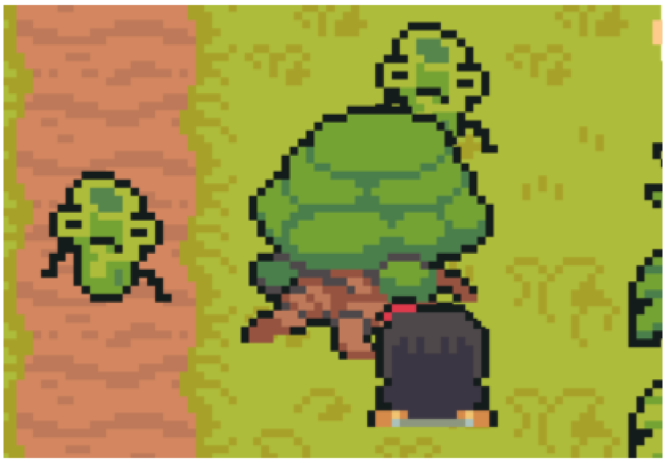
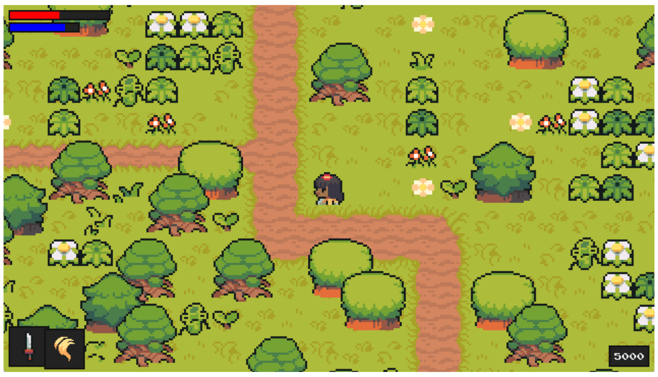
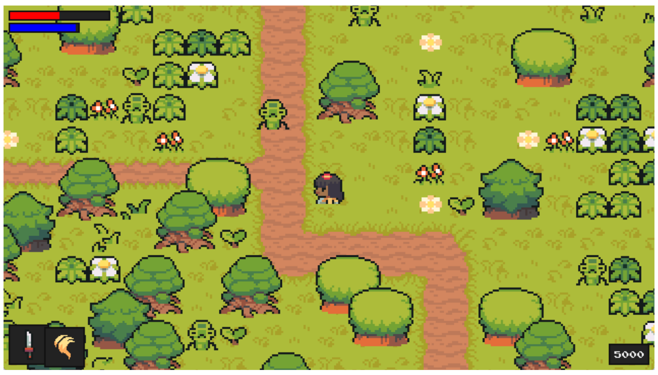
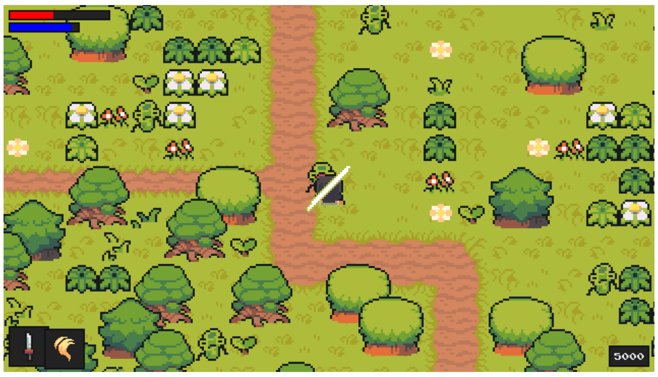
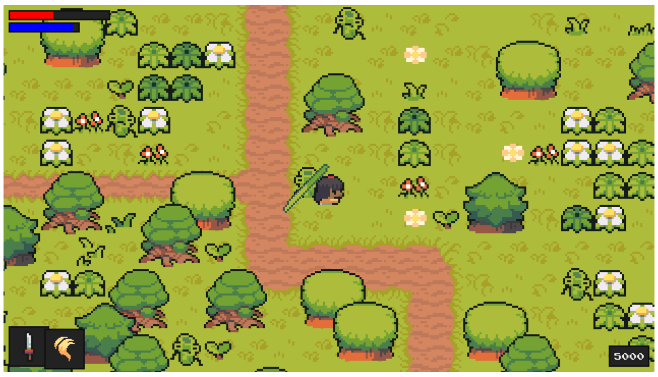


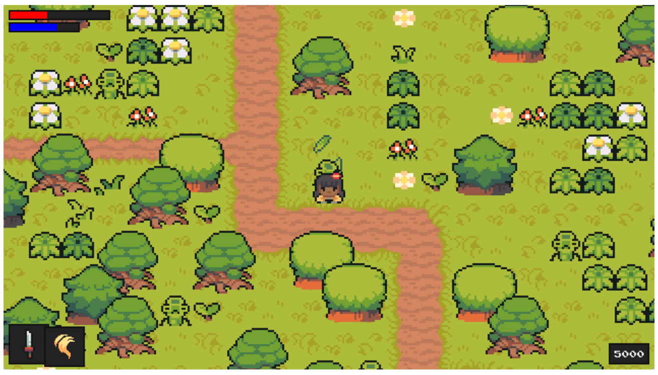
Figure 4-6: Enemies also have this obscuring effect



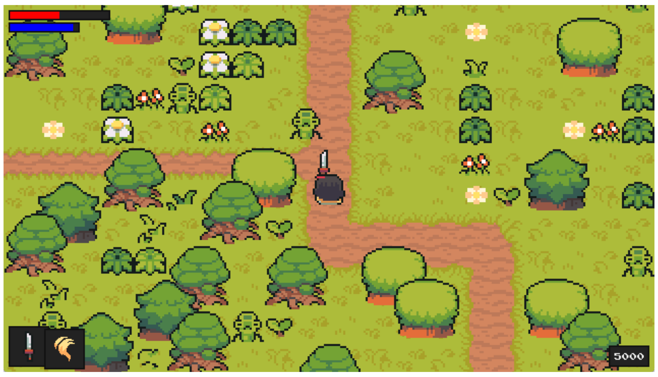


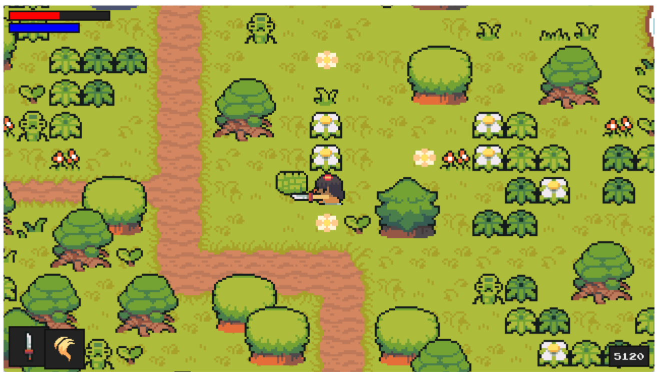


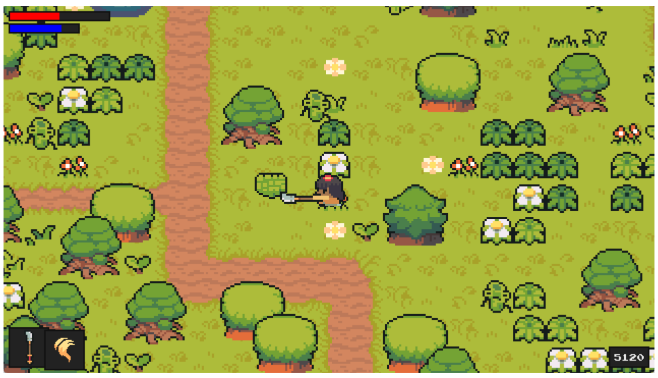




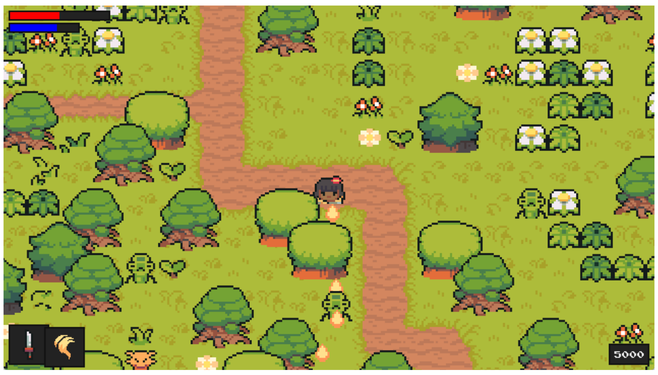
Figures 4-7 4-8 4-9 4-10 4-11: When far from the Enemy, the player is not detected; when the player gets closer, they are detected; when within attack range, the player will be attacked and the health bar will decrease; simultaneously, the character blinks, and during blinking, they are invincible; only when the invincibility status ends will the player receive damage again

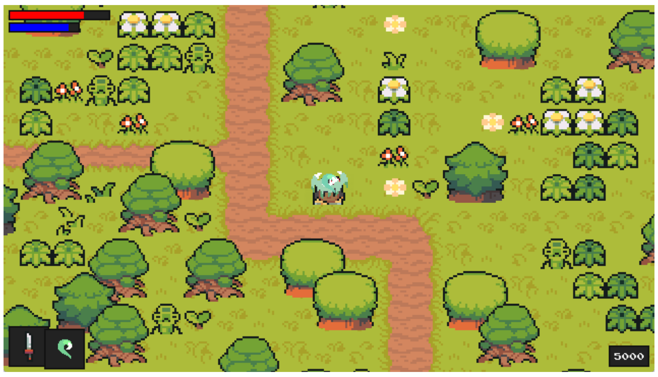






Figures 4-12 4-13 4-14: Player uses melee weapon to attack, monsters will be damaged and knocked back; if the monster's health reaches the kill line, a death animation will appear; different melee weapons can be used with different attack ranges and damages; the long sword in the image directly kills the Enemy





Figures 4-15 4-16: Player uses fire magic to attack, monsters will be damaged and knocked back, while consuming mana; healing magic can also be used to restore health by consuming mana

04 Reflection

In collision detection for obstacles, the approach used in this game is to load all obstacle information on the map and then perform collision detection for each obstacle; however, if the map is large, checking all obstacles during each update would be very time-consuming.

Adopting the approach we discussed in the Theory analysis, one method is to divide the map into sections. This way, when performing collision detection for obstacles, first determine the area where the player and enemy are located, and then check their collision with obstacles, which can reduce many unnecessary checks. Another approach is to classify obstacles by type, where certain obstacles only appear in specific areas, so collision detection is not needed when outside those areas.

In the collision detection between enemies and players, the game implementation similarly involves checking against all enemies on the map.

Here, a similar approach can be adopted as with handling obstacles, classifying monsters by region. However, since monsters move, changing their regions, we would also need to maintain data about the regions monsters are in.

Of course, noting the implementation of this game, it essentially creates a Camera functionality: the player is always at the center of the screen. Perhaps a detection method can be implemented with a possible collision range centered on the player, and only perform collision detection within this range.

Of course, in addition to the above improvements, many small design elements of this game are quite eye-catching, such as the implementation of visual occlusion effects or the visual effect of knockback, which transcend game interaction logic itself and elevate to the level of gaming experience. I have gained a lot!