Algorithms for Game Design

Collision Detection Algorithm in *DungeonRush* Game

Author: NoughtQ

School ID: 1145141919810

Date: 2024-12-13

2024-2025 Autumn&Winter Semester

Table of Contents

Chap 1: Introduction

1.1 Overview of the Game

DungeonRush is an open-source game designed by @rapiz1. The author said this game was inspired by the classic video game **Snake**, where the player maneuvers the head of a snake and keeps the snake from colliding with both other obstacles and itself, which is one of the main gamestyle of DungeonRush.

The programming languages used in the game is pure **C** with **SDL**, and the latter one, with the full name **S**imple **D**irectMedia **L**ayer, is a library providing a hardware abstraction layer for computer multimedia hardware components, often used to write high-performance computer games and other multimedia applications, and it can be included as C library.

Here is the main user interface of the game:

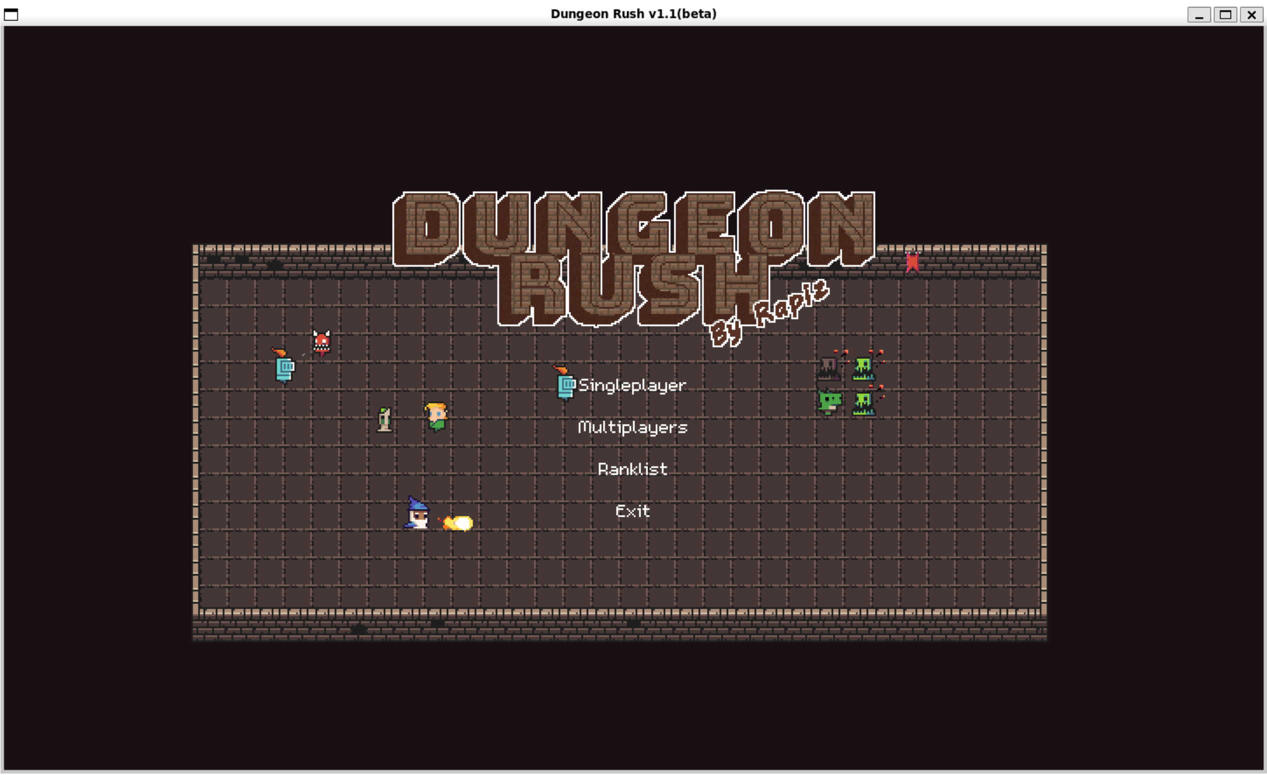

Figure 1: Main UI of the Game

There are four options in the main UI:

- **Singleplayer**: Start playing the game by yourself.
- **Multiplayers**: Provide two different ways to achieve multiplayers game.

- Local: Play the game with two players, when one person uses arrow keys to control, and the other person uses WASD to control.
- Lan: Play the game remotely with your friends in Local Area Network.

- **Ranklist**: List top 10 records in decreasing order by scores.
- **Exit**: Exit the game(Escape key has the same effect).

After selecting the Singleplayer mode or Local or Lan options in Multiplayers mode, we should choose the difficulty levels: **normal**, **hard** or **insane**, in increasing difficulty. Once we choose the appropriate difficulty, we finally start the game, and here is the snapshot of the game.

Figure 2: Snapshot of the Game(Marked)

In the snapshot, we can identify the major components of the games:

- **A randomly generated map**, with random shape and random distribution of sprites, items and so on.
- **Information board** (on the left upper corner of the window) recording the stage, the score of each player and "Find x more heores!", which is the necessary requirement to win the current stage of the game.

  - The game has "infinity"(but limited by the range of integer) stages, and with the stage number increasing, the game will be more chanllenging!
  - To **win** one stage of the game, player(s) should enlarge his(or their) queue(s)(called **snake**) to meet the number shown in the info board by approaching and collecting the heroes distributed randomly in the map, which is similar to the original Snake game.
- **Heroes**: Sprites in the game that player(s) should collect to win the stage. The game provides four types of heroes:
  - Knight: He has the most hp and his weapon is a sword, which has narrow attack range and medium damage, but quick attack.
  - Elf: His weapon is a bow, which has a relatively long range but small damage
  - Wizzard: He has the least hp and his weapon is the default magic called fireball, which has long shooting range, large damage, but slow attack.
  - Lizard: His weapon is his claw, which has a short but relatively-large range.
- **Monsters**: Enemies attacking players' snakes, and in fact **they are also snakes**. Because there are many kinds of monsters, I don't list them in my report.
- **Items**: Useful things for players, including:
  - Medicines: They can be used to recover the hp of the whole snake.
  - Weapons: Some monsters will drop the weapons after their death, and these weapons have stronger power than the default weapons of heroes.
- **Traps**: Spikes that can hurt the snake(s) at any time.

When the game is over, it will show the score of the game.

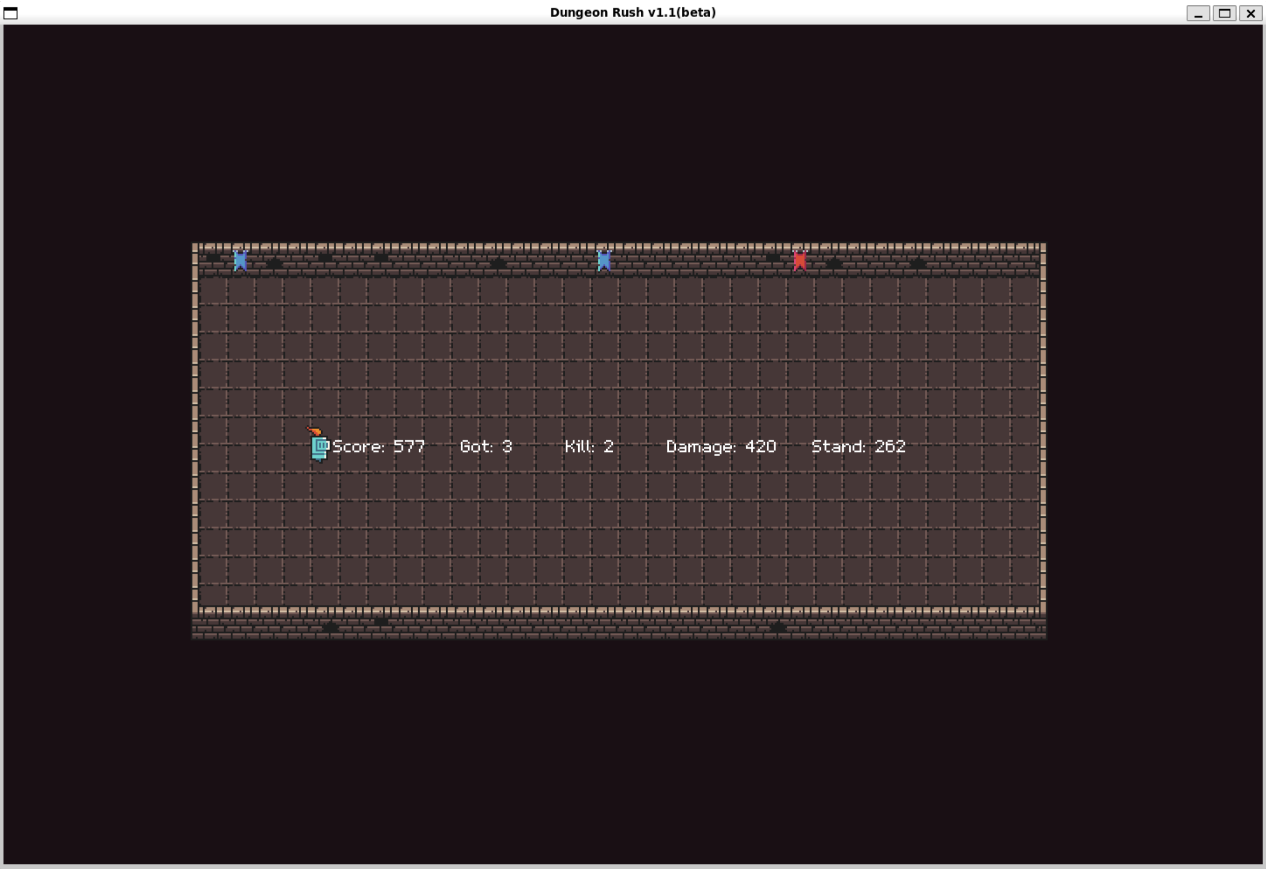

Figure 3: Score of the Game

- Score
- Got: the number of heroes we have collected.
- Kill: the number of monsters the snake killed.
- Damage: the sum of damage that the snake made.
- Stand: the sum of damage by monsters that the snake stood.

1.2 File Architecture of Source Codes

To gain a glimpse into the source codes of the game, let's take a look at the whole file architecture:

```
.
├── CmakeLists.txt                     // Compiling Tools
├── cmake
│   └── sdl2
├── res                                // Resources
│   ├── audio
│   ├── drawable      // Pictures of Sprites, Items and so forth
│   └── font
```

```
└── src                                   // Source Codes
    ├── adt.h           // ADT(specifically it's linked list)
    ├── ai.c            // algorithms about AI
    ├── ai.h
    ├── audio.c
    ├── audio.h
    ├── bullet.c        // Operations of bullets
    ├── bullet.h
    ├── game.c          // The main logic of the game
    ├── game.h
    ├── helper.c        // Helper functions
    ├── helper.h
    ├── main.c          // The main function
    ├── map.c
    ├── map.h
    ├── net.c           // Network
    ├── net.h
    ├── player.c        // Initialize and Create snakes
    ├── player.h
    ├── prng.c          // Pseudo random number generator
    ├── prng.h
    ├── render.c        // Animation
    ├── render.h
    ├── res.c           // Handling resources
    ├── res.h
    ├── sprite.c
    ├── sprite.h
    ├── storage.c       // Store the scores and ranklist
    ├── storage.h
    ├── text.h
    ├── types.c         // Misc
    ├── types.h
    ├── ui.c
    ├── ui.h
    ├── weapon.c
    └── weapon.h
```

1.3 Algorithm Description

Among dozens of algorithms used in the game, in this report I will introduce the collision detection algorithms used in the game in detail. The reasons for choosing this kind of algorithms are:

- **Importance to the game**: In DungeonRush game, there are many collision cases, such as the collision between sprites and monsters, bullets, items and wall block, which is a little complicated but it's worthwhile to analyzing them for its importance to the game.
- **Relevance to the course**: In the course *Algorithm for Game Design*, Professor William Nace have taught us the algorithms of collision detection in the previous lecture, and I had a general but shallow understanding of this kind of algorithms. To gain a deeper insight into collision detection algorithms, I'm passionate about learning these algorithms used in the game.

Now let's have a look at collision detction algorithms in general.

- **Collision** is one of the major interaction between sprites in games. To handle the collisions, we should detect these collisions first. As a consequence, collision detection algorithms are one of the core algorithms used in the game.
- Instead of detecting each pixel of sprites(it's too expensive and impractical!), we consider the sprites as **rectangles** (called rects as shortcut). So in each iteration of game loop, we should only check the conflictions between corresponding rects.
- In order to correctly and effciently detect and handle the collisions, algorithm desiners should consider these important aspects or strategies:
  - **Reduce** the number of checks that need to be made
    1. Don't care offscreen objects
    2. Partition objects(quadtree, BSP, ...)
  - **Quickly test** sprites (pairwise): discard those that can't possibly collide
    1. Circle/Sphere test
    2. Bounding box test
    3. Capsule test
  - **Thoroughly test** remaining sprites to detect collisions
    1. Swept sphere algorithm

- **Resolve** collisions: Do whatever – explosions, bounces, etc.

Chap 2: Algorithm Explanation

> Note
>
> I have given the theoretical description on the general collision algorithms in Chapter 1, so I won't repeat it again.

In this chapter, I will explain the specific collsion detection and resolution algorithms applied in the game in theoretical and implementing perspecitve.

You should find a function called `makeSnakeAttack()` and `makeCross()` in the path `./src/game.c`, which are the main collision algorithms responsible for **active collisions** and **passive collisions** respectively.

In the meantime, you may also find two subroutines called `makeSnakeCross()` and `makeBulletCross()` in function `makeCross()` , detecting and resolving the collision of snakes(queues of heroes) and bullets(for gun-like weapons by both heroes and monsters) respectively. And in `makeSnakeCross()`, `crushVerdict()` is an important subroutine used to resolve the fatal crush.

However, these functions are mainly in charge of collision resolution. In fact, the pure collision detection algorithms is `RectRectCross()`, which we will introduce it first.

I have organized the logic of these collision algorithms in DungeonRush into pseudo-codes, which is more streamlined than the source codes and it' more convenient for us to understand their essence.

2.1 Collision Detection

2.1.1 RectRectCross()

Here is the pesudo-code of this function:

> Note
>
> In the source codes, this function is implemented by three subfunctions. But I integrate them into one function for considering them as a whole.

Procedure: rectRectCross(a: **SDL_Rect**, b: **SDL_Rect**)

Begin
  $al \leftarrow a{\rightarrow}x$
  $ar \leftarrow a{\rightarrow}x + a{\rightarrow}w$
  $at \leftarrow a{\rightarrow}y$
  $ab \leftarrow a{\rightarrow}y + a{\rightarrow}h$

  $bl \leftarrow b{\rightarrow}x$
  $br \leftarrow b{\rightarrow}x + b{\rightarrow}w$
  $bt \leftarrow b{\rightarrow}y$
  $bb \leftarrow b{\rightarrow}y + b{\rightarrow}h$

  $intervalX \leftarrow$ max(min(ar, br) - max(al, bl), 0)
  $intervalY \leftarrow$ max(min(ab, bb) - max(at, bt), 0)
  $interval \leftarrow intervalX \cdot intervalY$
  return $interval >= RECT\_CROSS\_LIMIT$
End

1. Calculate the left, right, top and bottom **border** of the rect of a and b.
2. Calculate the **overlapped area** (*interval* in the pseudo-code) in two rects.
3. Return the result whether the overlapped area exceeds the **limitation**.

The following picture shows the algorithm in a graphic and vivid way:

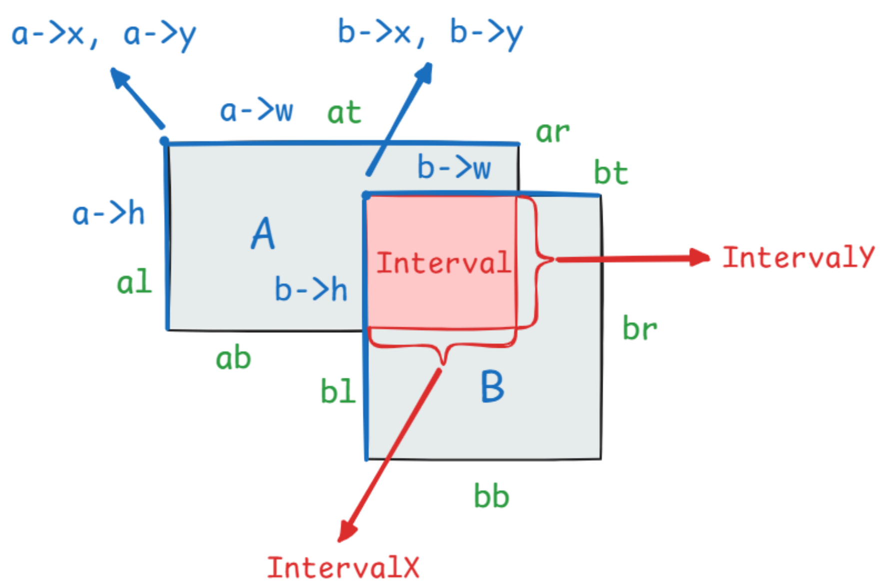

Figure 4: Diagram of the RectRectCross() algorithm

2.2 Collision Resolution

2.2.1 makeSnakeAttack()

> Note
>
> In the source codes, this function is implemented with a subfunction called `makeSpriteAttack()` for handling the attack from each sprite. For the similar reasons as shown in `RectRectCross()` algorithm, the pseudo-code below inserts the whole implementation of `makeSpriteAttack()` into its parent routine.

```
Procedure: makeSnakeAttack(snake: Snake struct)
Begin
    if snake is frozen then return
    for sprite in snakes do
        weapon ← sprite's weapon
        attacked ← false

        for all characters in the map do
            if the character is not in snake then
                consider it as the possible target
                if the distance between sprite and target is larger than
                the shootrange of weapon then continue
                attacked ← true

                if weapon isn't a gun-like weapon then
                    render the animation of effect of weapon
                    damage target
                    if possible, add the debuff to target
                else
                    create the bullet of the weapon
                    render the animation of effect of weapon
                endif

                attacked ← true
```

```
                if weapon can't attack multiple targets then get out
                from the loop
            endif
        end

        if attacked then
            render the audio and animation of the attack effect
        endif
    end
End
```

Here are the critical steps to make attack from sprites in the snake:

1. Check if the snake is frozen. If not, continue executing following steps.
2. Check every sprite in the snake

   1) Except his teammates, the sprite considers all characters as his possible damage target.

   2) Check if the distance between the sprite and the target is larger than the shootrange of sprite's weapon. If yes, continue to find the next possible target, otherwise run the following steps

   3) Resolve the damage effect of sword-like and gun-like weapon respectively

   4) Check if the weapon can attack multiple targets. If not, check the next sprite, otherwise continue to find the next target.

2.2.2 makeCross()

Here is the pesudo-code.

```
Procedure: makeCross()
Begin
    for snake in snakes do
        makeSnakeCross(snake)
    end
    for bullet in bullets do
        if makeBulletCross(bullet) then
            remove the animation of bullet
            remove bullet from bullets
```

```
        endif
    end
End
```

As you see, the main logic and operations of collision detection algorithms are hiden in subroutines `makeSnakeCross()` and `makeBulletCross()` actually, so we need to learn algorithms from them further.

2.2.3 makeSnakeCross()

According to the convention, pseudo-code is given first:

```
Procedure: makeSnakeCross(snake: Snake struct)
Begin
    if no sprite in snake then
        return false
    endif

    for every block in the window do
        if the block is in the map then
            if the block is a trap block and the trap is enabled then
                for sprite in snake do
                    if sprite touches the trap then
                        sprite is damaged by the trap
                    endif
                end
            endif

        if RectRectCross(rect of snake's head, block) then
            if there is a hero on the block then
                append the hero to snake
                remove the animation of the original hero from the block
            else if there is a hp-medicine on the block then
                take up the medicine and recover some hp for all sprites
                in snake
```

```
                remove the animation of the original medicine from the block
            else if there is a weapon on the block then
                the first appropriate sprite in snake takes up this weapon, or no sprite can take up it.
                if the weapon is taken then
                    remove the animation of the original weapon from the block
                endif
            endif
        endif
    end

    for sprite in snake do
        if sprite's hp is equal to 0 then
            record the death situation
            play the audio and animation about sprites' death
            remove the died sprite from snake
        endif
    end

    Update survived sprites' position in snake
    if no sprite in snake then
        return false
    endif

    die ← crushVerdict(snake)
    if die then
        kill all sprites in snake
    endif

    return die
End
```

The logic of the algorithm is very clear, but to have a comprehensive understanding of this algorithm, we can divide the whole algorithm in several stages:

1. Check if the snake is **empty** (died). If not, then continue the following steps.
2. Check all **special blocks** in the map, and here are four cases we should consider(other cases should be ignored):
    - **Traps**: if the trap is enabled(i.e. the spikes are appeared in the picture), then spikes touching the spikes are gotten hurted.
    - **Heroes**: append heroes to the snake(‘s tail)
    - **Hp-medicines**: recover all survived sprites’ hp values in the snake.
    - **Weapons**: the first appropriate hero can take the weapon(often more powerful then the original one).
3. Check all sprites’ **hp values** in the snake. If the hp value is zero, then remove the died sprite from the snake. Afterwards, the positions of survived sprites should updated in time.
4. Check if the snake **crushes** on wall blocks or itself. If it’s in the case, then unfortunately all the snake will be dead.
5. **Return a boolean value** indicating whether the snake has survived in the collision.

2.2.4 crushVerdict()

> Note
>
> To facilitate explanation, I properly modify the parameters of the algorithm without affecting the analysis of algorithm logic.

The corresponding pseudo-code is also shown first:

Procedure: crushVerdict(*sprite*: **Sprite struct**)

```
Begin
    for all blocks around sprite(a 3 × 3 space) do
        if the block is out of the map and RectRectCross(block, rect of sprite)then
            return true
        endif
    end
```

```

    for snake in snakes do
        for other_sprite in snake do
            if RectRectCross(rect of other_sprite, rect of sprite)then
                return true
            endif
        end
    end

    return false
End
```

The algorithm considers two cases that have fatal damage to the snake:

- one of the sprites in the snake crashes on the **wall blocks(borders)**.
- one of the sprites in the snake crashes on **other snakes** (including itself).

2.2.5 makeBulletCross()

The corresponding pseudo-code is also shown first:

```
Procedure: makeBulletCross(bullet: Bullet struct)
Begin
    get the rect of bullet
    find the weapon of bullet

    if bullet is outside of the map then
        regard bullet as a hit
        render the animation of hit effect
    endif

    if not hit then
        for snake in snakes do
            if bullet not belongs to snake then
                for sprite in snake
                    if RectRectCross(rect of sprite, rect of bullet) then
                        render the animation of hit effect
```

```
                        reduce the hp value of sprite
                        add the debuff on sprite(if it has)
                        return hit
                    endif
                end
            endif
        end
    endif

    if hit then
        render the animation of effect by weapon
        for snake in snakes do
            for sprite in snake
                if sprite is in the damage range of the weapon then
                    reduce the hp value of sprite
                    add the debuff on sprite(if it has)
                endif
            end
        end
    endif

    return hit
End
```

The steps of this algorithm is shown below:

1. Get the **necessary info** of bullet, for following collision detection and resolution.
2. Check if the bullet is **in the map**. If not, consider it as a hit.
3. Check if the bullet **hits any sprites** (including all heroes and monsters, except the owner snake of the bullet) existing in the map. If it's in the case, then damage the sprite, may add a debuff to the sprite in the meantime and return hit info immediately.
4. Check **the damage of large-effect-range weapon** for every sprites. If the sprite is in the range, then it will be damaged by the effect.
5. **Return the hit info**.

2.3 Complexity Analysis

> Note
>
> We only focus on the space and time complexity analysis on the collision detection and resolution algorithms above.

2.3.1 Space Complexity

Conclusion: $O(N + W \cdot H)$, where

- N: Number of existing characters in the map.
- W: Width of the screen
- H: Height of the screen

Analysis:

- Apparently, we should store the information for every character surviving in the game, including its hp, type, weapon and so forth, for detecting and resolving the collisions of them, and updating their status.
- Because the random shape and distributed map is limited to the game screen, and each block on the map needs to be stored to determine the collision between sprites and the blocks(may contains special items like medicines and weapons). Therefore, the space complexity of collision algorithms are also dominated by the size(width $\times$ height) of map.

In summary, the space complexity is $O(N + W \cdot H)$

2.3.2 Time Complexity

Conclusion: $O(T^2 \cdot M^2 + T \cdot M \cdot (W \cdot H + B))$, where

- T: Number of snakes
- M: Maximum number of sprites of the snake in all snakes
- W: Width of the screen
- H: Height of the screen
- B: Number of bullets

Analysis:

- `RectRectCross()`: It calculates the overlapped area of two rects of characters or items by info of the position of upper left point, width and height. So it only takes constant time to finish the calculations.
- `makeSnakeAttack()`:
  - The outermost loop of the algorithm is dominated by the number of snakes(T).
  - Then, we should check every sprite in the snake, and we take the maximum number of sprites of the snake in all snakes(M).
  - For each sprite, we should examine if other sprites not belonging to the currently checked snake are in the shootrange of the weapon($T \cdot M$).
  - Consequently, the overall time complexity of the algorithm is $O(T^2 \cdot M^2)$
- `makeCross()`: The algorithm call `makeSnakeCross()` function for all snakes(T) and `makeBulletCross()` function for all bullets(B).
  - `makeSnakeCross()`: For everyone in the snake, the algorithm should detect and resolve the collisions between the sprite and all blocks in the map(sprites number × width × height, i.e. $M \cdot W \cdot H$). Then it also handles the collisions between the head of the snake with other sprites on the map($T \cdot M$). As a result, the time complexity of this algorithm is $O(T \cdot M \cdot (W \cdot H + T))$
  - `makeBulletCross()`: For each bullet, the algorithm will detect and resolve the collision between the bullet and all characters in the map. So the time complexity is $O(T \cdot M)$

In a nutshell, the time complexity is $O(T^2 \cdot M^2 + T \cdot M \cdot (W \cdot H + B))$.

Chap 3: Execution and Observations

3.1 Executions

There are two ways to set up the game:

1. Simple method(for common users)

- Download the released package in the GitHub repository(link)
- Unzip the compressed package and go into the folder.
- You will find an executable file called `dungeon_rush.exe`(in Windows environment, no postfix in Linux environment). Open it and you should start the game successfully!

2. More complex way(for developers)

- Clone the repository to your local machine(using command `git clone`)
- Go into the folder, and you will find similar file architecture shown in Section 1.2.
- Open the terminal in this directory, input the command `cmake -B build && cmake --build build`(so you should install cmake tool first), wait a minite and then you will get a folder called `build`
- Go into the `build` folder, then go into the `bin` sub-folder. You will find an executable file. Just open it, and you are in the game!

3.2 Observations

Because it's impossible to demonstrate animations of the game in the report, I have no alternative but to list some interesting outcomes in text with some pictures.

- After increasing the constant `HELPER_RECT_CROSS_LIMIT` to a large number(the maximum overlapped area to be tolerated; collision happens when exceeding this constant),
  - even if sprites cross each other, nothing happens as the picture shows below!
  - More interestingly, all sprites can get out of the map owing to no collision with borders.

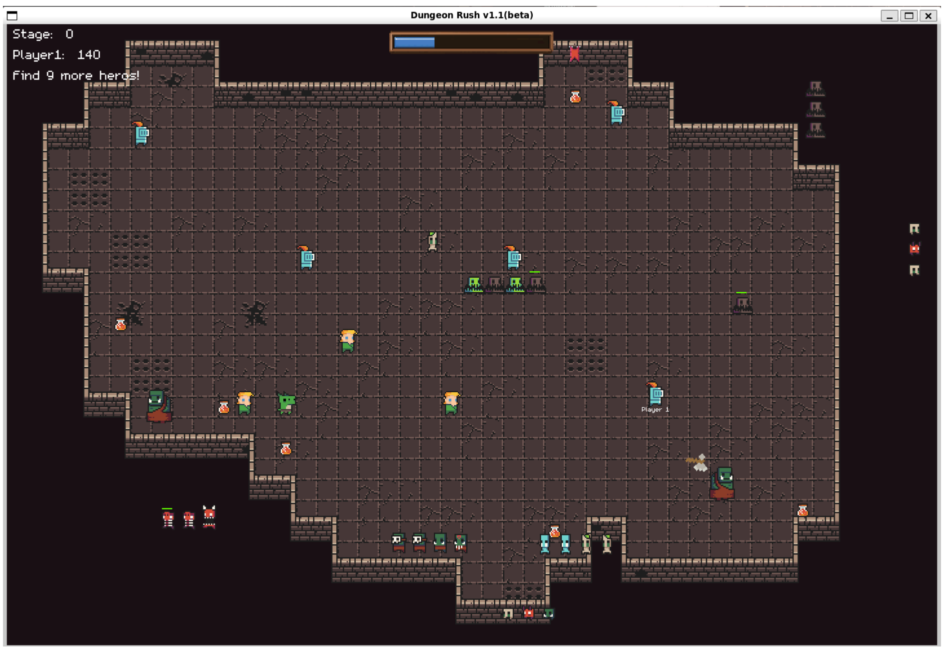

Figure 5: `HELPER_RECT_CROSS_LIMIT` is increased

- When stopping `gameLoop()` from calling subfunction `makeSnakeAttack()`, no sprite make attack to other snakes. As you can see in the following picture, even though heroes and monsters are in the shootrange of their weapons, neither of them make attack to one another, hence eliminating the active collisions.

Figure 6: `makeSnakeAttack()` is banned

- I also make attempt to invalidate the death caused by `crushVerdict()`. As you can see in the picture below, the snake of heores can go out of the map without death, and the monsters are still in the map.

Figure 7: no death when going across the wall

Chap 4: Reflection

4.1 Acquisition

I have benefited a lot from this lab, and in the following section I will summarize my gain from these aspects.

- **Details of collision detections**: In DungeonRush game, collision detection is implemented by comparing the overlapped area of two rectangles of sprites with the preset limitation, which was also mentioned in the previous course. Despite its simplicity, I have learned the specific implementation of this kind of collision detection in C langauge, which gives me a strong inspiration to develop my own algorithm.
- **Thorough consideration of all cases of collisions**: As the report has shown above, there are tons of collision cases should be detected and resolved, so the corresponding algorithms will be complicated and lengthy. In spite of the difficulty, the necessary requirement is taking all possible cases into account, and algorithms in the game make it! So this is a good example and opportunity for me to learn that awareness.

4.2 Potential Improvements

Although *DungeonRush* is an awesome game, and the source-codes are run correctly, it may still have some flaws and have the space to improve. Here are my suggestions:

- **More comments for comprehension**: Maybe coding in this project isn't a so chanllenging job for the author, so the source codes are lack of comments. As a consequence, it has an bad influence on reading codes fluently for me. From my perspecitve, writing comments is a fabulous habit for project design and implementation.
- **More well-organized codes**: In chapter 2, although I have split the collision algorithms into detection and resolution parts, actually functions like `makeSnakeAttack()` and `makeSnakeCross()` combine collision detection with resolution. In my opinion, when designing a complete algorithm, we should divide the algorithm into several submodules according to their functions, for better understanding and the possible modifications subsequently.