Documentação do jogo

Hugo Alves Azevedo de Souza
Rafael Moisés de Sá Tavares

Introdução

O jogo desenvolvido como trabalho final da disciplina de Algoritmos e Estrutura de
Dados I foi baseado no jogo clássico de Atari Berzerk, sendo esse feito a partir de um modelo
fornecido pelo professor em um repositório do GitHub. O Jogo foi feito em 2D e tem como
objetivo vencer uma sequência de fases que levam a um chefão. O projeto tem uma tela de
início que pega um nickname digitado pelo jogador, com o intuito de criar um placar. Após
essa tela, tem outra que serve para selecionar a dificuldade, sendo ela duas opções: normal ou
difícil. Após isso, surge o herói controlado pelo jogador, o qual tem 2 tiros disponíveis no
modo normal e 1 no difícil, para matar os inimigos que também possuem tiro com
quantidades opostas ao herói, ou seja, 1 no normal e 2 no difícil. Cada mapa tem uma
configuração diferente. Todos têm barreiras diferentes e quantidades de inimigos diferentes.
Caso morra para inimigos, o mapa será reiniciado, como se o jogador tivesse acabado de
entrar. Além disso, tem um chefão final que possui um comportamento diferente em relação
aos inimigos padrões. Ele sempre possui dois tiros, independentemente do modo e a cada tiro

que leva, ele reduz o seu tamanho, e quando leva 10 ao todo, ele se divide em 10 inimigos
que atiram também, tendo a quantidade de bala ditada pelo modo de jogo escolhido no início.
Caso o herói seja morto pelo chefão, será game over. Se o herói matar o boss, ele irá vencer o
jogo e irá aparecer automaticamente um placar assim que matar todos inimigos. O placar tem
uma mensagem de vitória e mostra as pontuações das últimas três rodadas. Essas informações
são guardadas em um arquivo chamado “pontuacoes.txt” e serão mostradas no placar
somente se o jogador matar o boss. Caso o jogador sair do jogo antes da fase final, ou morra
na parte do boss, a pontuação será desconsiderada. Essa pontuação salva no arquivo e
mostrada no placar é o tempo em segundos que o herói levou para sair da primeira fase até
matar o boss. O tempo gasto nas duas telas de início não é contado.

Tiro do herói e inimigos

Para implementar o tiro no herói e nos inimigos, foi utilizado como base a estrutura bullet e a função update_bullet. A estrutura contém todos os atributos da bala e a função atualiza a posição da bala no mapa quando ela é acionada. Abaixo se encontra a declaração da struct Bullet:

```
// src/game/game.h
typedef struct Bullet{
    Rectangle pos;
    Rectangle default_pos;
    Color color;
```

```
    int speed;
    int active;
    int direction;
}Bullet;
```

Como pode ser observado a struct tem 7 atributos necessários para criar uma bala no jogo, sendo pos a posição atual da bala no mapa; default_pos uma posição em que a bala é escondida quando não está sendo utilizada; color, representando a cor do projétil; speed que constitui a velocidade da bala; active que auxilia na identificação se a bala está em andamento ou não; e por último, o atributo direction onde é armazenado a direção do projétil. Todos os personagens do jogo possuem duas variáveis do tipo Bullet em suas estruturas, possibilitando a eles efetuar no máximo dois disparos.

Sempre que um tiro é disparado a bala recebe a posição atual do personagem que efetuou o disparo e tem seu estado active modificado para o valor 1. Assim, quando a função update_bullet é chamada, será passado para ela três parâmetros, dois ponteiros do tipo Bullet, que representam a bala um e a bala dois do personagem, e o ponteiro do jogo que é do tipo Game.

```
// src/bullet/bullet.c
void update_bullet(Bullet *b, Bullet *b2, Game *g) {
    if(b->active){
        if(b->direction == KEY_LEFT) {
                b->pos.x -= b->speed;
            if (!(b->pos.x > SCREEN_BORDER)) {
                b->active = 0;
                b->pos = b->default_pos;
            }
            if(barrier_collision(&g->maps[g->curr_map],&b->pos)){
                b->active = 0;
                b->pos = b->default_pos;
            }
```

```
        }
        if(b->direction == KEY_RIGHT) {
            b->pos.x += b->speed;
            if (!(b->pos.x + b->pos.width < g->screenWidth -
SCREEN_BORDER)){
                b->active = 0;
                b->pos = b->default_pos;
            }
            if(barrier_collision(&g->maps[g->curr_map],&b->pos)){
                b->active = 0;
                b->pos = b->default_pos;
            }
        }
        if(b->direction == KEY_UP) {
            b->pos.y -= b->speed;
            if (!(b->pos.y > SCREEN_BORDER)) {
                b->active = 0;
                b->pos = b->default_pos;
            }
            if(barrier_collision(&g->maps[g->curr_map],&b->pos)){
                b->active = 0;
                b->pos = b->default_pos;
            }
        }
        if(b->direction == KEY_DOWN) {
            b->pos.y += b->speed;
            if (!(b->pos.y +b->pos.height < g->screenHeight -
SCREEN_BORDER)){
                b->active = 0;
                b->pos = b->default_pos;
            }
            if(barrier_collision(&g->maps[g->curr_map],&b->pos)){
                b->active = 0;
                b->pos = b->default_pos;
            }
        }

    }
      . . .
}
```

Nesse trecho da função é mostrado apenas as operações realizadas com o primeiro ponteiro Bullet recebido pela função, porém o procedimento é o mesmo para a variável b2 que também é recebida no parâmetro da função. Sendo assim é possível observar que a função só atualiza a posição do projétil se o estado active estiver com um valor diferente de 0, e a atualização da posição da bala ocorre de acordo com a direção do tiro, em que a posição x e y do projétil é somada ou subtraída como valor da velocidade da bala, e quando a mesma colidir com alguma barreira ela é escondida e o estado active passa a ser 0.

Agora que entendemos como a structure Bullet e a função update_bullet funciona, veremos como elas são implementadas em cada personagem.

- O tiro do herói:

  Conforme já mencionado, na struct do heróis há duas variáveis do tipo Bullet, o que possibilita a ele efetuar até dois disparos:

```
// src/game/game.h
typedef struct Hero
{
    Rectangle pos;
    Color color;
    int speed;
    int special;
    int kill;
    Bullet bullet;
    Bullet bullet2;
} Hero;
```

  Além disso, o herói atira sempre que a tecla de espaço é pressionada e a direção do disparo será a direção em que o herói estava se movimentando. Portanto na função update_hero_pos, é feito uma verificação para atribuir a

atual direção do herói em suas balas caso elas não tenham sido disparadas.

```
void update_hero_pos(Game *g)
{
    Hero *h = &g->hero;
    Map *m = &g->maps[g->curr_map];
    Bullet *b = &h->bullet;
    Bullet *b2 = &h->bullet2;
    if (IsKeyDown(KEY_A) || IsKeyDown(KEY_LEFT))
    {
        if (h->pos.x > SCREEN_BORDER)
            h->pos.x -= h->speed;
        if (barrier_collision(m, &h->pos))
            h->pos.x += h->speed;
        (b->active == 0) ? b->direction = KEY_LEFT: b->direction;
        (b2->active == 0) ? b2->direction = KEY_LEFT:
b2->direction;

    }
      ...
```

Em seguida, na função UpdateGame sempre é chamado a função shoot que tem como papel, chamar a função update_bullet passando como parâmetro os ponteiros das balas do herói, e também verificar se a tecla de espaço foi pressionada. Caso essa verificação seja verdadeira e o herói não esteja com aquela bala ativada, a posição da bala será atualizada para a posição do herói e a bala passará a ter o atributo active igual a 1, simbolizando que aquela bala está em andamento. Também é possível notar que a bala 2 do herói só é ativada se o modo de jogo for o modo normal.

```
// src/bullet/bullet.c
void shoot(Hero *b, Rectangle *position, Game *g) {
    update_bullet(&b->bullet, &b->bullet2,g);

    if(IsKeyPressed(KEY_SPACE) &&  b->bullet.active == 0){
```

```
        if(b->bullet.direction == KEY_UP || b->bullet.direction
== KEY_DOWN) {
            b->bullet.pos =
(Rectangle){position->x,position->y,15,45}; //15, 45
        }
        if(b->bullet.direction == KEY_LEFT || b->bullet.direction
== KEY_RIGHT) {
            b->bullet.pos =
(Rectangle){position->x,position->y,45,15};
        }
        b->bullet.active = 1;
    }
    else if(IsKeyPressed(KEY_SPACE) &&  b->bullet2.active == 0 &&
(g->mode == 'N')){
        if(b->bullet2.direction == KEY_UP || b->bullet2.direction
== KEY_DOWN) {
            b->bullet2.pos =
(Rectangle){position->x,position->y,15,45};
        }
        if(b->bullet2.direction == KEY_LEFT ||
b->bullet2.direction == KEY_RIGHT) {
            b->bullet2.pos =
(Rectangle){position->x,position->y,45,15};
        }
        b->bullet2.active = 1;
    }
}
```

Para a colisão do tiro com os inimigos é verificado se as balas do herói colidiu com a posição do inimigo, e caso tal verificação seja verdadeira, a bala é escondida e desativada e o inimigo não é mais desenhado. Caso o inimigo em questão tenha a chave para a porta do próximo mapa, a porta será liberada.

```
// função Update_Game em src/game/game.c
if (CheckCollisionRecs(g->hero.bullet.pos, map->enemies[i].pos))
        {
            map->enemies[i].draw_enemy = 0;
            map->enemies[i].pos = (Rectangle){4500,4000,1,1};
```

```
            map->enemies[i].enemyBullet.pos =
map->enemies[i].enemyBullet.default_pos;
            map->enemies[i].enemyBullet.active = 0;
            map->enemies[i].enemyBullet2.pos =
map->enemies[i].enemyBullet2.default_pos;
            map->enemies[i].enemyBullet2.active = 0;
            g->hero.bullet.active = 0;
            g->hero.bullet.pos = g->hero.bullet.default_pos;
            if (map->enemies[i].has_key)
            {
                map->door_locked = 0;
            }
        }

if (CheckCollisionRecs(g->hero.bullet2.pos, map->enemies[i].pos))
        {
            map->enemies[i].draw_enemy = 0;
            map->enemies[i].pos = (Rectangle){4500,4000,1,1};
            map->enemies[i].enemyBullet.pos =
map->enemies[i].enemyBullet.default_pos;
            map->enemies[i].enemyBullet.active = 0;
            map->enemies[i].enemyBullet2.pos =
map->enemies[i].enemyBullet2.default_pos;
            map->enemies[i].enemyBullet2.active = 0;
            g->hero.bullet2.active = 0;
            g->hero.bullet2.pos = g->hero.bullet2.default_pos;
            if (map->enemies[i].has_key)
            {
                map->door_locked = 0;
            }
        }
```

- Tiro do inimigo e do boss:

  O tiro do inimigo e do boss funciona de maneira similar ao do herói, exceto pelo fato que o boss sempre tem dois tiros independentemente do modo de jogo e o disparo dos tiros dos inimigos são feitos em relação à posição do herói.

```
// função update_enemy_pos em src/enemy/enemy.c
update_bullet(&e->enemyBullet, &e->enemyBullet2, g);
```

```
    if(!e->enemyBullet.active && rand() % 5 == 1){
        int x = g->hero.pos.x - e->pos.x;
        int y = g->hero.pos.y - e->pos.y;

        if(x>=-10 && x<=10){
            e->enemyBullet.direction = (y>0) ? KEY_DOWN : KEY_UP;
            e->enemyBullet.pos = (Rectangle){e->pos.x, e->pos.y,
15, 45};
            e->enemyBullet.active = 1;
        }
        if(y>=-10 && y<=10){
            e->enemyBullet.direction = (x>0) ? KEY_RIGHT :
KEY_LEFT;
            e->enemyBullet.pos = (Rectangle){e->pos.x, e->pos.y,
45, 15};
            e->enemyBullet.active = 1;
        }
    }
    else if(!e->enemyBullet2.active  &&  rand() % 50 == 1 &&
(g->mode == 'H')){

        int xx = g->hero.pos.x - e->pos.x;
        int yy = g->hero.pos.y - e->pos.y;

        if(xx>=-10 && xx<=10){
            e->enemyBullet2.direction = (yy>0) ? KEY_DOWN :
KEY_UP;
            e->enemyBullet2.pos = (Rectangle){e->pos.x, e->pos.y,
15, 45};
            e->enemyBullet2.active = 1;
        }
        if(yy>=-10 && yy<=10){
            e->enemyBullet2.direction = (xx>0) ? KEY_RIGHT :
KEY_LEFT;
            e->enemyBullet2.pos = (Rectangle){e->pos.x, e->pos.y,
45, 15};
            e->enemyBullet2.active = 1;
        }
    }
```

Na função update_enemy_pos sempre é chamada a função update_bullet para caso aquele inimigo específico tiver uma bala ativa, a posição do projétil será atualizada. Nessa mesma função é feito uma comparação da posição do inimigo e do herói e caso o herói esteja em uma posição em que o X ou o Y dele sejá próxima do X ou Y daquele inimigo (ou seja, os dois estão na mesma faixa horizontal ou vertical), o inimigo tem uma chance de efetuar um disparo em direção ao herói. Com o boss o sistema é praticamente o mesmo.

Caso a bala do herói e dos inimigos colida, ambas são desativadas e escondidas do mapa.

```
// src/game/game.c na função UpdateGame
bulletCollison(&map->enemies[i].enemyBullet, &g->hero.bullet);
bulletCollison(&map->enemies[i].enemyBullet2, &g->hero.bullet);
bulletCollison(&map->enemies[i].enemyBullet2, &g->hero.bullet2);
bulletCollison(&map->enemies[i].enemyBullet, &g->hero.bullet2);

// src/bullet/bullet.c
void bulletCollison(Bullet *b1, Bullet *b2){
    if(CheckCollisionRecs(b1->pos, b2->pos)){
        b1->active = 0;
        b2->active = 0;
        b1->pos = b1->default_pos;
        b2->pos = b2->default_pos;
    }
}
```

O herói morre quando é colidido com a bala de um inimigo e o mapa em que ele está é restaurado.

```
// stc/game/game.c na função UpdateGame
if (map->enemies[i].enemyBullet.active &&
CheckCollisionRecs(map->enemies[i].enemyBullet.pos, g->hero.pos)){
                map->enemies[i].enemyBullet.pos =
map->enemies[i].enemyBullet.default_pos;
                if(!g->hero.special) {
                    if(g->curr_map == 8)
```

```
                    g->gameover = 1;
                    resetMap(g);
                    continue;
                }
            }

            if (map->enemies[i].enemyBullet2.active &&
CheckCollisionRecs(map->enemies[i].enemyBullet2.pos, g->hero.pos) &&
(g->mode == 'H')){
                map->enemies[i].enemyBullet2.pos =
map->enemies[i].enemyBullet2.default_pos;
                if(!g->hero.special) {
                    if(g->curr_map == 8)
                    g->gameover = 1;
                    resetMap(g);
                    continue;
                }
            }
```

Implementação dos mapas

Em todos os mapas exceto o do chefão há um item que faz com que o herói fique invulnerável aos tiros dos inimigos e possibilita a ele matar os inimigos quando colidir com eles.

O mapa 1 possui apenas uma barreira no centro da tela que ocupa quase todo eixo Y, os inimigos dessa fase são extremamente lentos porém suas balas são mais rápidas para compensar a lentidão dos inimigos.

O mapa 2 possui apenas duas barreiras e 6 inimigos. Os inimigos dessa fase são muito rápidos, porém seus tiros são extremamente devagar o que faz a fase ser muito caótica porém possível de passar.

O mapa 3 possui três barreiras e também 6 inimigos, porém com velocidade moderada sendo a velocidade do inimigo e das balas deles quase a mesma.

O mapa 4 possui cinco barreiras, sendo três na parte superior do mapa e duas na parte inferior, esse mapa possui

oito inimigos extremamente lentos, tendo a própria velocidade reduzida e de suas balas também.

O mapa 5 possui três barreiras no centro da tela, esse mapa possui 8 inimigos que são rápidos e possui balas mais rápidas ainda.

O mapa 6 tem quatro barreiras e nesse mapa os inimigos são mais lentos do que os inimigos da fase anterior mas ainda possuem balas muito rápidas.

O mapa 7 possui apenas dois blocos como barreira e inimigos que tem velocidade moderada, porém a segunda bala do inimigo é mais rápida do que a primeira.

O mapa 8 tem 5 barreiras e seus inimigos também possuem velocidade moderada, mas tem bala mais lentas do que suas próprias velocidades.

E por último o mapa final, possui o boss que tem 10 vidas e quando morre é dividido em 10 inimigos.

Conclusão

As partes ligadas ao funcionamento e jogabilidade foram todas implementadas, assim

como requisitado. O trabalho foi concluído totalmente. Os requisitos cumpridos foram:

● A criação de uma tela inicial para escrever o nome e escolher o modo de jogo;

● Possui 8 fases normais, com inimigos e mapas diferentes, e uma fase final, com um

chefão diferente dos demais inimigos, totalizando 9 fases;

● O herói tem uma aparência diferente das dos inimigos, possuem direções para

deslocar e tem o tiro feito pela tecla “espaço”, sendo esse tendo a quantidade de

munição disponível de acordo com o modo;

● Os inimigos e o herói estão com o tiro implementado e funcionando, ambos
funcionando adequadamente. Se o tiro acertar em algum, esse morre;
● O herói morre se levar tiro e o inimigo também. A colisão do jogo está funcional.
Tiro com tiro, tiro com herói, tiro com inimigo, tiro com barreiras e bordas do mapa;
● O código está todo separado e estruturado em módulos bem definidos, para facilitar a
leitura e a codificação. Toda a parte ligada a funções e struct está em um “.h” e cada
parte do jogo, como barreiras, herói, inimigo, tiro, mapa, etc., estão cada um
separados em “.c”s;
● Foi implementado dois níveis: normal e difícil;
●
● Possui um placar que mostra as últimas pontuações do jogador. A pontuação é salva
em um arquivo e só é colocada nele caso o jogador ganhe o jogo;
● Possui uma documentação que especifica o jogo criado.
Em relação à parte extra, não foi feita nenhuma parte dela. Não tem música, sprites ou
um placar que mostra as 3 maiores pontuações.
As maiores dificuldades foram na implementação do placar, na parte de salvar e ler as
informações para usá-las de maneira adequada, a criação de um placar que nunca
enchesse e só guardasse as 4 mais recentes. Além disso, outra dificuldade foi o toque
inicial com o jogo. A quantidade de informação que tinha que entender e manipular

foi uma dificuldade. Entender como que cada coisa funciona e como manipular isso

inicialmente foi mais difícil que implementar qualquer outra coisa do jogo, fora o

placar. A partir disso, foi mais fácil. A criação do vilão, por exemplo, e as

funcionalidades dele foram fáceis, pois já estávamos acostumados com o jogo.