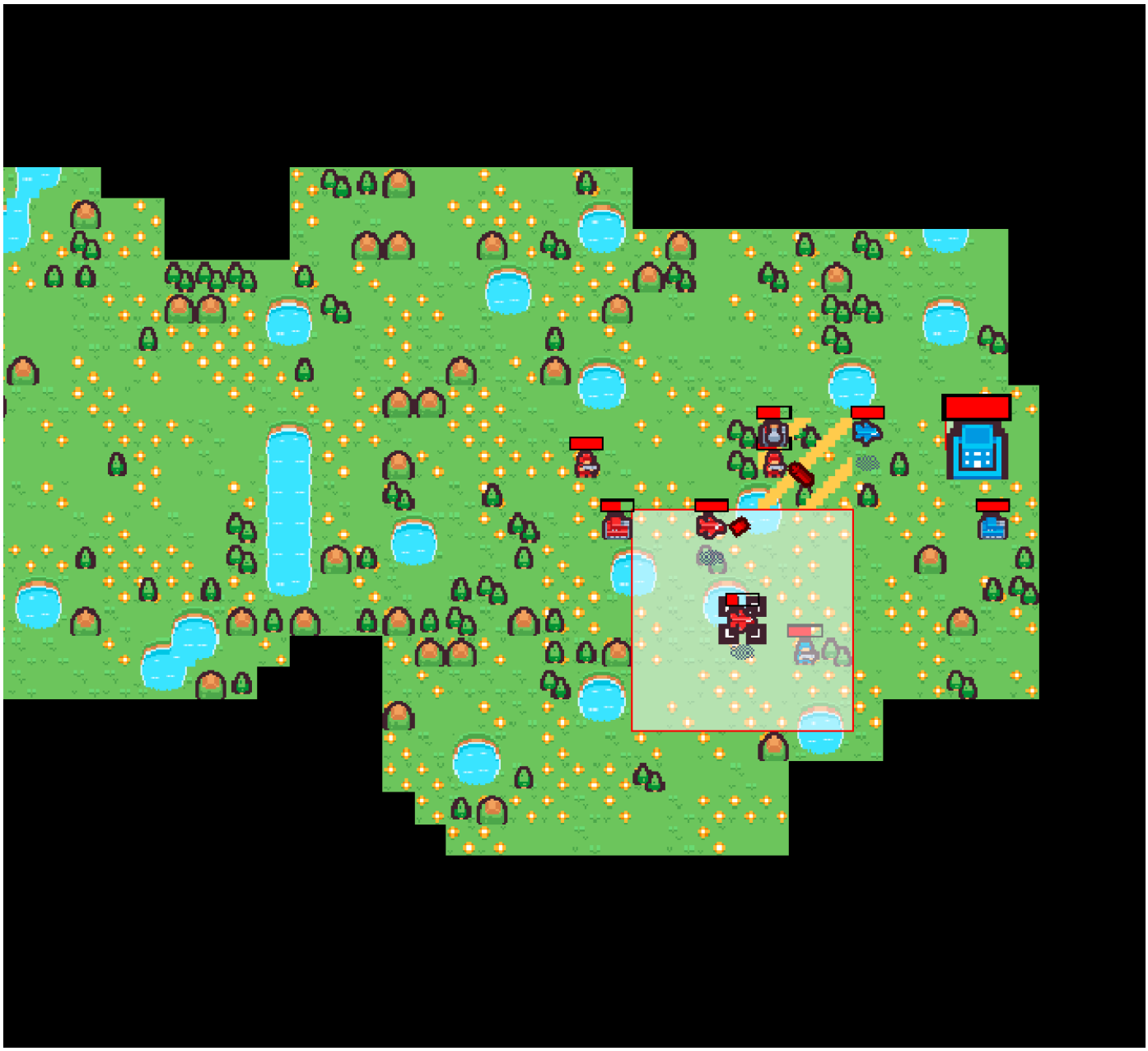
一、介绍

这是一个基于SFML的c++2D游戏，实现了地图的创建，四类不同的地形，三种不同的士兵和各自的移动方式与攻击方式。提供了方便的人工控制方式，包括：移动、攻击、切换控制角色、显示攻击范围、显示控制角色高量框、生成士兵、呼叫己方士兵、调整视图等功能。另外还实现了电脑控制角色，包括：自动移动、自动攻击、发现敌人时追击、自动生成士兵等行为。



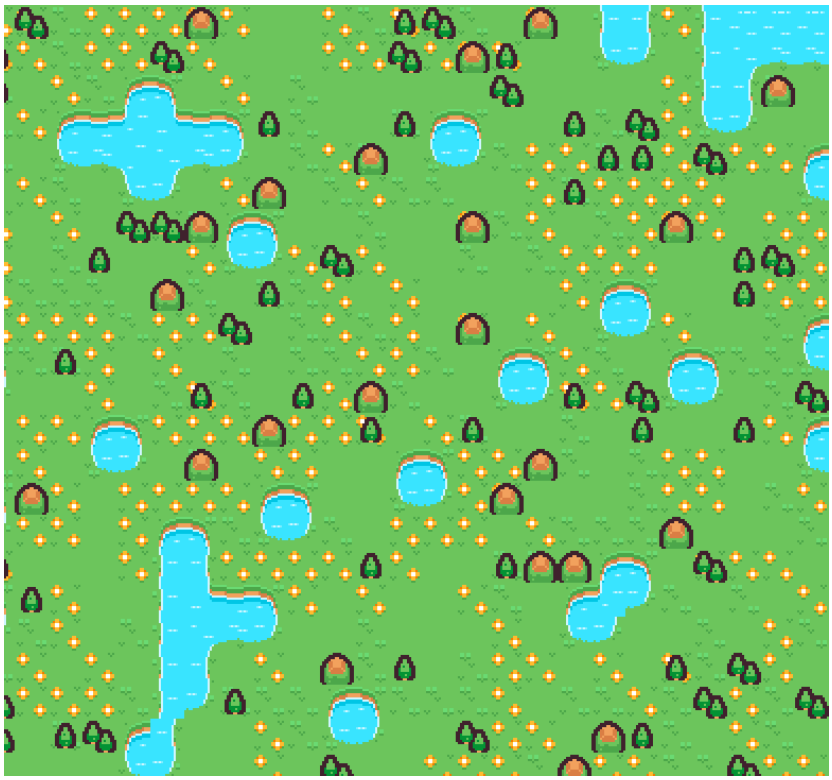
二、安装与运行

安装SFML库之后可以在src目录下执行`../bin/game`命令来运行游戏。

三、游戏特点

1. 合理的水域贴图

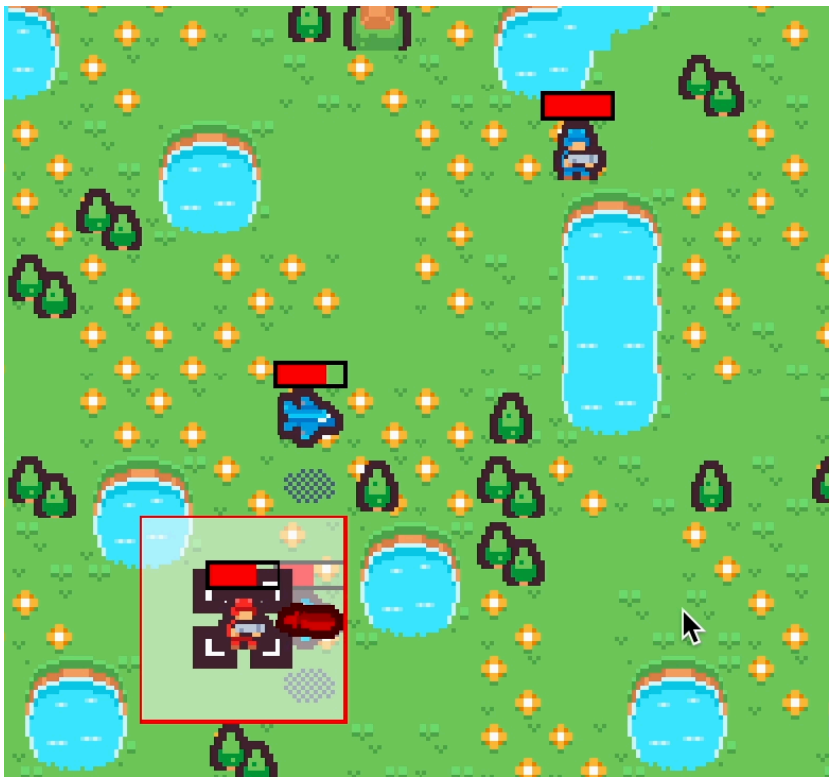
属性为水的格子会判断相邻格子是水或是陆地，并选择合适的纹理，这样就可以生成合理美观的水域。



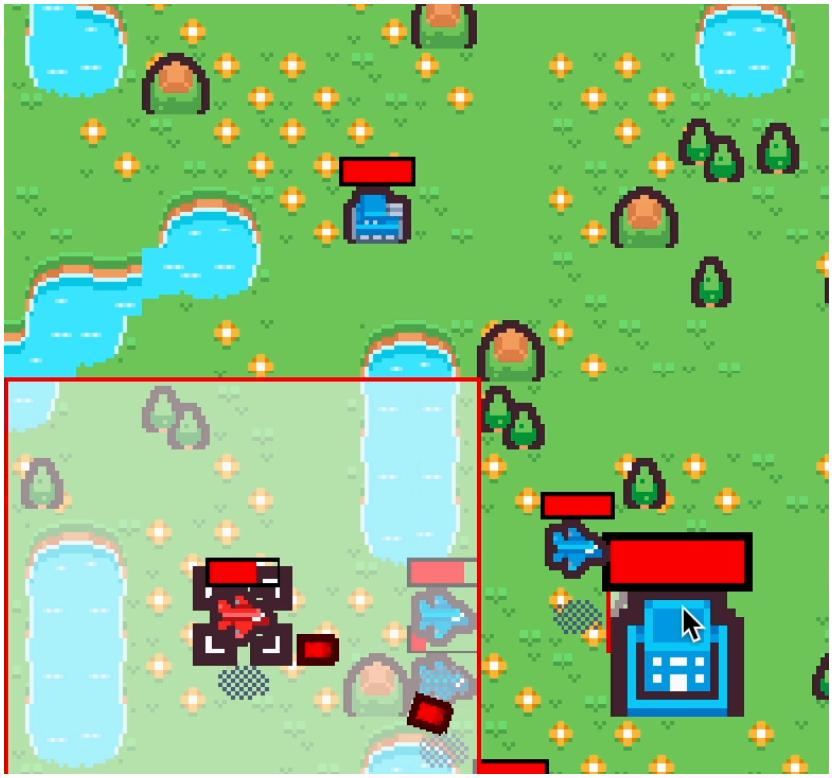
2. 多种地形、多种士兵、多种移动方式和攻击方式

如上图，游戏实现了多种地形：平地、水、山、树林。

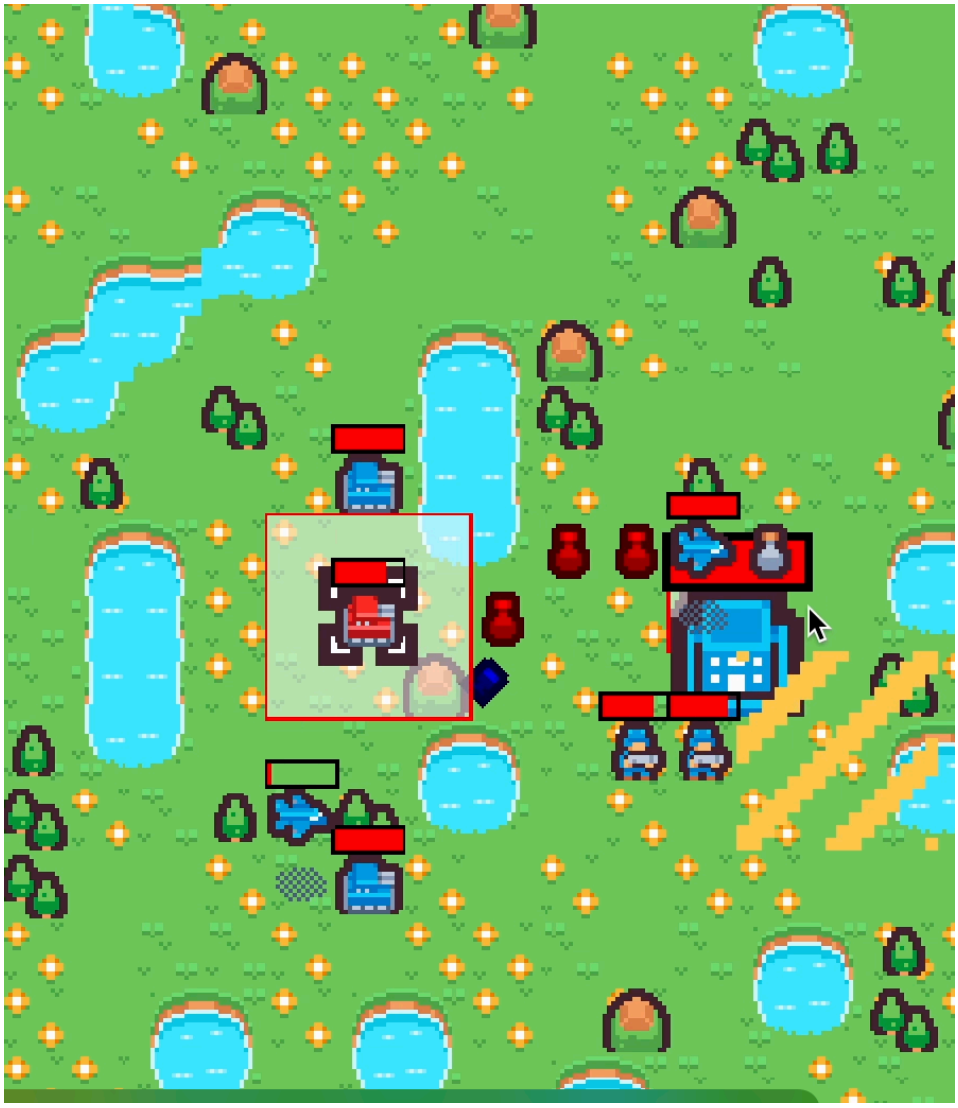
此外实现了三种士兵：战士、飞机、坦克。他们的移动方式分别为走、飞、开车。他们的攻击方式为剑、子弹、炸弹。



战士的移动方式是走，可以通过除了山和水的地形。攻击方式是剑，范围是一格范围内最近的敌人。



飞机的移动方式是飞，可以通过所有地形。攻击方式是子弹，范围是三格内最近的敌人。



坦克的移动方式是开车，只可以通过平地。攻击方式是炸弹，会在原地安放一个炸弹，2.5秒后爆炸，对一格范围内的所有敌人都造成伤害。

所有的攻击方式都制作了攻击动画。

3.方便的操作方式

游戏有很多便于玩家操作的设计，包括当前操作角色的高量框显示，攻击范围显示，鼠标所指方格显示等。也提供了很多操作方式，包括wasd移动，空格键攻击，tab切换控制角色，\键打开关闭战争迷雾，]键控制攻击范围显示，[键控制高量框显示，‘键控制视角的锁定。当视角不锁定时可以使用方向键移动视角。可以使用- +键控制视角的缩放。

另外还提供了一些特色的操作：

1. 按下鼠标使当前控制角色以自己的移动方式按最短路径移动到鼠标按下时所在位置。（战争迷雾笼罩的地方将被视为不可通过）
2. 控制任意士兵单位按下e键会生成一个号令之旗，会召唤所有友军来到旗帜所在地。（旗帜是有动画的）

四、实现细节

1.整体框架

mian函数中只负责接收玩家的输入和显示，在这里初始化GAME类。

```cpp
int main()
{
    //窗口设置
    sf::RenderWindow window(sf::VideoMode(720,450),"MY GAME!");
    window.setFramerateLimit(60);
    //游戏初始化
    GAME game;
    //视图控制
    sf::View view(sf::FloatRect(0,0,720,450));
    //主循环
    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))…
        //显示
        window.clear();
        if(game.player1->view_focused)
        {
            view.setCenter(game.player1->focus->get_position());
        }
        window.setView(view);
        window.draw(game);
        window.display();
    }
    return 0;
}
```

GAME类中包含地图与各个玩家，可以方便地拓展到多个玩家（不论是人类控制还是电脑控制）

```
113 ////////////////////GAME//////////////////////////
114 //          游戏类，包含了所有的玩家和地图              //
115 ////////////////////////////////////////////////
116 class GAME : public sf::Drawable, public sf::Transformable
117 {
118 private:
119
120 public:
121     bool game_end_flag;
122     //地图加载
123     MAP* map;
124     //文本显示
125     sf::Font font;
126     sf::Text text;
127     sf::RectangleShape box;
128     //鼠标框显示
129     destinationBox* destination_box;
130     //玩家
131     PLAYER* player1;
132     //人机
133     PLAYER* computer;
134     GAME();
135     ~GAME();
136     void draw(sf::RenderTarget &target, sf::RenderStates states) c
137     void onof_fog_of_war();
138     void onof_view_focus();
```

PLAYER类，包含自己的战争迷雾，和自己的单位。

```
//////////////////PLAYER//////////////////////////
//       玩家类，目前提供两个阵营：红色和蓝色          //
////////////////////////////////////////////////
class PLAYER : public sf::Drawable, public sf::Transformable
{
private:
    // 私有成员变量和函数在这里声明

public:
    // 公共成员变量
    GAME* game;
    MAP* players_map;
    char color;
    std::vector<live_things*> things;
    FOG* fog_of_war;
    bool show_fog_of_war;
    FOCUS* focus;
    bool view_focused;
    bool thread_running;
```

Live_things类，这是一个基类，各种士兵是继承于此基类。其中移动与攻击通过组合的方式实现。

```
class live_things : public sf::Drawable, public sf::Transformable
{
public:
    // 纹理和精灵，用于绘制对象
    sf::Texture texture;
    sf::Sprite sprite;

    // 行为控制器，用于控制移动和攻击行为
    move_behavior* move_behav;
    attack_behavior* attack_behav;

    // 视野范围
    int field_of_vision;

    // 颜色标识
    char color;

    // 所属的玩家
    PLAYER* belong_to;

    // 是否由AI控制
    bool AI_controlled;

```

```
    // 线程运行状态
    bool thread_running;

    // 是否已经死亡
    bool is_dead;

    // 健康条
    mutable HealthBar health_bar;

    // 当前生命值
    int HP;

    // 目标路径
    std::stack<direction> destination_path;

    // 构造函数
    live_things();

    // AI控制函数
    virtual void ai_control();

    // 关闭线程函数
    void shut_thread();

    // 绘制函数
    virtual void draw(sf::RenderTarget &target, sf::RenderStates states) const;

    // 照亮迷雾函数
    sf::Rect<int> light_up_fog();

    // 获取位置函数
    sf::Vector2f get_position();
    sf::Vector2i get_position_in_map();

    // 死亡函数
    virtual void die();

    // 在地图上移动函数
    void move_in_map(direction d);

    // 工厂函数，用于创建新的live_things对象
    virtual live_things* factory(live_things_type type);

    // 析构函数
    virtual ~live_things();

    // 受到伤害函数
    void take_damage(int damage);

```

```
    // 设置目标位置函数
    void set_destination(sf::Vector2i destination);
};
```

focus类，这是player类中提供给外界控制palyer中单位的接口。

```
//////////////////FOCUS//////////////////////////
//     用于手动操控角色，提供一些操控的接口            //
////////////////////////////////////////////////
class FOCUS: public sf::Drawable, public sf::Transformable
{
private:
    live_things* focus_thing;//当前focus的角色
    highlightBox* focus_box;//高亮框
    atackRangeBox* atack_range_box;//攻击范围框
    bool show_focus_box;
    bool show_atack_range_box;
public:
    FOCUS();
    void set_focus(live_things* focus);//设置focus
    void draw(sf::RenderTarget &target, sf::RenderStates states) const;
    void onof_focus_box();//显示/关闭高亮框
    void onof_atack_range_box();//显示/关闭攻击范围框
    void move_focus(direction d);//移动focus
    void change_focus();//切换focus，切换到下一个角色
    void set_destination(sf::Vector2i destination);//设置目的地，然后开始BFS寻路
    void on_aicontrol();//开启电脑控制
    void attack();//攻击
    void factory(live_things_type type);//生产单位
    live_things* get_focus_thing();
    sf::Vector2f get_position();
};
```

2.重要细节

单位的电脑控制，使用多线程实现

```
void live_things::ai_control()
{
    // 使用多线程实现
    std::thread t([this](){
        thread_running=true; // 线程开始运行
        bool path_done=true; // 路径完成标志
        while(thread_running&&!is_dead) // 当线程运行且对象未死亡时
        {
            while(thread_running&&AI_controlled&&!is_dead) // 当线程运行、AI控制且对象未死亡时
            {
```

```
                live_things* target=this->attack_behav->find_target(); // 寻找目标
                if(target!=NULL) // 如果找到目标
                {
                    this->attack_behav->atack(); // 攻击目标
                    std::this_thread::sleep_for(std::chrono::milliseconds(2000));
// 休眠2秒
                    continue;
                }
                else if(this->destination_path.size()>0) // 如果有目标路径
                {
                    path_done=false; // 路径未完成
                    direction d=this->destination_path.top(); // 获取路径顶部的方向
                    this->destination_path.pop(); // 弹出路径顶部的方向
                    this->move_in_map(d); // 在地图上移动
                    std::this_thread::sleep_for(std::chrono::milliseconds(500)); //
休眠0.5秒
                }
                else
                {
                    path_done=true; // 路径完成
                    live_things* temp=this->attack_behav->see_target(); // 看到目标
                    if(temp!=NULL) // 如果看到目标
                    {
                        this->set_destination(temp->get_position_in_map()); // 设置
目标位置
                        path_done=false; // 路径未完成
                        continue;
                    }
                    direction d=static_cast<direction>(rand()%4); // 随机选择一个方向
                    this->move_in_map(d); // 在地图上移动
                    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
// 休眠1秒
                }
            }
            // 当AI_controlled为false时，清空destination_path，防止一次路径没有走完时由切
换到手动控制造成的bug
            if(!path_done)
            {
                while(!destination_path.empty()) // 清空目标路径
                {
                    destination_path.pop();
                }
                path_done=true; // 路径完成
            }
            std::this_thread::sleep_for(std::chrono::milliseconds(1000)); // 休眠1秒
        }
    });
    t.detach(); // 分离线程，让它自行运行
}
```

寻路算法

使用BFS找到最短路径，使用了一个二位数组（类似floyed算法中的prev数组）存放前驱节点。使用队列实现广度优先搜索，使用栈存放路径。

```
void live_things::set_destination(sf::Vector2i destination)
{
    // 获取起始位置
    sf::Vector2i start=this->get_position_in_map();

    // 获取迷雾信息和地图信息
    bool* fog_of_war=this->belong_to->fog_of_war->get_fog_of_war();
    node* map=this->belong_to->players_map->get_map();

    // 使用BFS算法，寻找最短路径
    bool visited[MAP_WIDTH][MAP_HEIGHT]; // 访问标记数组
    sf::Vector2i prev[MAP_WIDTH][MAP_HEIGHT]; // 前驱节点数组

    // 初始化访问标记数组
    for(int i=0;i<MAP_WIDTH;i++)
    {
        for(int j=0;j<MAP_HEIGHT;j++)
        {
            visited[i][j]=false;
        }
    }

    // BFS队列
    std::queue<sf::Vector2i> q;
    q.push(start);
    visited[start.x][start.y]=true;

    while(!q.empty())
    {
        sf::Vector2i current=q.front();
        q.pop();

        // 如果到达目标位置，结束搜索
        if(current==destination)
        {
            break;
        }

        // 检查四个方向
        sf::Vector2i directions[4] = {sf::Vector2i(0, -1), sf::Vector2i(0, 1),
sf::Vector2i(-1, 0), sf::Vector2i(1, 0)};
        for(int i = 0; i < 4; i++)
        {
            sf::Vector2i next = current + directions[i];
```

```
44
45              // 如果下一个位置在地图内，未被访问，可通行，且不在迷雾中
46              if(next.x >= 0 && next.x < MAP_WIDTH && next.y >= 0 && next.y <
    MAP_HEIGHT && !visited[next.x][next.y] && this->move_behav->passable(map, next) &&
    !fog_of_war[next.x+next.y*MAP_WIDTH])
47              {
48                  q.push(next);
49                  visited[next.x][next.y]=true;
50                  prev[next.x][next.y]=current;
51              }
52          }
53      }
54
55      // 如果目标位置未被访问，说明无法到达目标位置
56      if(!visited[destination.x][destination.y])
57      {
58          return;
59      }
60
61      // 重构路径
62      for(sf::Vector2i pos = destination; pos != start; pos = prev[pos.x][pos.y])
63      {
64          direction d;
65          if(pos.x-prev[pos.x][pos.y].x==1)
66          {
67              d=RIGHT;
68          }
69          else if(pos.x-prev[pos.x][pos.y].x==-1)
70          {
71              d=LEFT;
72          }
73          else if(pos.y-prev[pos.x][pos.y].y==1)
74          {
75              d=DOWN;
76          }
77          else if(pos.y-prev[pos.x][pos.y].y==-1)
78          {
79              d=UP;
80          }
81
82          // 将方向压入目标路径栈
83          destination_path.push(d);
84      }
85  }
```

攻击动画的实现

使用多线程实现动画的更新

```
void attack_knife::atack()
{
    live_things* target=this->find_target();
    if(target!=NULL&&!target->is_dead)
    {
        target->take_damage(this->ATK);
        std::thread animation_thread(&attack_knife::attack_animation, this,
target);
        animation_thread.detach();
    }
}
```

以剑为例，动画的实现如下。另外考虑到玩家输入时攻击频率可能会很快，使用队列的方式，每发起一次攻击，就新建一个剑的精灵并入队，动画结束就出队，使用智能指针的方式来避免内存泄露。

```
        // 根据目标在自身的八个方向，设置剑的位置和方向
        {…
        // 设置剑的角度和位置
        (*sprite).setRotation(angle-45);
        (*sprite).setPosition(knife_pos);
        // 旋转剑的角度以实现动画效果
        int count=0;
        while(count<30)
        {
            if(!show_atk)
                break;
            count++;
            (*sprite).rotate(3);
            std::this_thread::sleep_for(std::chrono::milliseconds(10));
        }
    }

    // 动画结束，移除精灵
    if(atk_object_sprites.size()>0)
        atk_object_sprites.pop();
        sprite=nullptr;
}
```

```cpp
void attack_knife::attack_animation(live_things* target)
{
    // 创建一个新的精灵用于显示攻击动画
    std::shared_ptr<sf::Sprite> sprite=std::make_shared<sf::Sprite>();
    atk_object_sprites.push(sprite);
    *sprite=atk_object_sprite;
```

五、效果展示

见视频

六、总结与未来改进方向

1. ---TODO---将纹理存在内存中，避免重复加载
2. ---DONE---实现对象的攻击、死亡。(爆炸？)
3. ---DONE---创建坦克、飞机类，继承自live_things，实现不同的移动方式
4. ---DONE---将HOME设置为工厂，可以生产步兵、坦克、飞机
5. ---DONE---增加鼠标，实现点击切换focus，导航
6. ---TODO---实现金币系统，实现购买功能
7. ---DONE---实现电脑控制非focus的角色，多线程，实现多个角色同时移动，实现电脑控制的player
8. ---TODO---考虑树木的可破坏性，但是这可能涉及到地图访问的冲突
9. ---TODO---考虑将地图的生成和加载分开，生成地图后保存到文件，加载地图时从文件读取
10. ---HALFDONE--考虑实现移动、攻击的动画，添加攻击范围显示在移动时改变纹理的方向
11. ---DONE---增加开始、结束动画
12. ---TODO--对项目结构、格式的优化，添加注释，解决循环包含的问题，改善函数、变量的调用，内存问题，使用const &这种方式
13. ---DONE---需要对地图进行优化，考虑使用结构体或者类的数组来存储地图信息，将士兵的位置也存储在地图中

```
(END)...skipping...
* c25ab24 (HEAD -> main) 最终版本
* 4895c9f 实现了攻击范围的显示，并且重构了focus
* 7badf6d 实现了号令之旗，下面考虑修改focus的机制
* b8783be 实现了鼠标方块显示，以及鼠标点击开始导航的功能，使用的是BFS
* 255c50f 实现了血量和死亡，但是HOME的死亡有问题，死亡后还可以factory一次
* 6d72c8b 将全局变量改成了用指针访问，实现了地图中存储士兵的位置信息
* ff0794f 初始版本，地图为全局数组。下个版本考虑改为类或结构体数组，要能存储士兵的位置。
```