



**本科毕业设计（论文）**

**决战象棋游戏开发**

**学 院 应用数学学院**

**专 业 信息与计算科学（计算方向）**

**年级班别 2014级（2）班**

**学 号 3114008162**

**学生姓名 梁浩翔**

**指导教师 刘志煌**

**2018年06月**

**中文摘要**

象棋起源于中国，最早可追溯到战国时期。随着时代的变迁，象棋的个数和种类以及玩法都有所变化，以至于演变成现在我们熟知的中国象棋。象棋的演变和创新可以让更多人参与到益智类竞技当中。

决战象棋是一款基于中国象棋行走规则的带有战斗力比较的游戏，当棋子攻击敌方棋子时会比较双方属性，胜者留，败者亡。结合战斗力比较以改变传统的中国象棋着法思维模式，让游戏带有不一样的趣味和挑战性，让双方玩家在游戏互动中得到更好的思维锻炼。

本系统的创新依赖于计算机的辅助计算，利用Unity 3D游戏引擎开发，分为5个模块：计时模块、复盘模块、战斗模块、资源重用模块和界面。

**关键词：**战斗计算，复盘，资源重用， Unity 3D

**Abstract**

Chess originate in China, it dates back to the warring states period. With the changing times, the number and type of chess and the playing method make a difference, which evolves Chinese Chess weknow.The evolution and innovation of chess can help more people participate in the competitive sports.

Battle chess, based on the rules of ChineseChess, is a game with comparison of combat power. It will compare the attributes of both sides when the chess attacks the enemy chess, winner survived, loser gone.Combiningcomparison of combat power to change traditional Chinese chess thinking mode, which makes the game fun and challenging, and makes both player get better mentai exercise in the game.

Innovation of the present system relies on computer assisted calculations .This system uses Unity 3D engine development, includeingfive modules: battle,time, chessboard reuming, resource reuse, and interface.

**Key words:** battle calculation,chessboard resuming,resource reuse,Unity3D

**目录**

[1 诸论 1](#_Toc514624228)

[1.1 研究背景与意义 1](#_Toc514624229)

[1.2 决战象棋设计研究方法 1](#_Toc514624230)

[1.3 本文的主要工作 1](#_Toc514624231)

[1.4 本文的结构 1](#_Toc514624232)

[2 系统的分析和设计 2](#_Toc514624233)

[2.1 需求分析 2](#_Toc514624234)

[2.1.1 界面需求 2](#_Toc514624235)

[2.1.2 非功能性需求 4](#_Toc514624236)

[2.2 开发所需条件和相关技术 4](#_Toc514624237)

[2.3 总体设计 4](#_Toc514624238)

[2.4 数据结构 5](#_Toc514624239)

[2.5 资源重用 5](#_Toc514624240)

[2.6 战斗属性处理 6](#_Toc514624241)

[3 系统实现 7](#_Toc514624242)

[3.1 主体框架实现 7](#_Toc514624243)

[3.1.1 资源重用实现 7](#_Toc514624244)

[3.1.2 数据层实现 8](#_Toc514624245)

[3.1.3 游戏控制器的实现 10](#_Toc514624246)

[3.2 界面实现 12](#_Toc514624247)

[3.3 棋盘表示 18](#_Toc514624248)

[3.4 棋子实现 20](#_Toc514624249)

[3.4.1 共有属性和行为 20](#_Toc514624250)

[3.4.2 各兵种行走规则 24](#_Toc514624251)

[3.5 战斗实现 39](#_Toc514624252)

[3.6 时间管理 42](#_Toc514624253)

[3.7 悔棋和复盘 44](#_Toc514624254)

[4 总结 49](#_Toc514624255)

[4.1 总结 49](#_Toc514624256)

[4.2 技术难点与解决 49](#_Toc514624257)

[4.3 展望 50](#_Toc514624258)

[参考文献 52](#_Toc514624259)

[致谢 53](#_Toc514624260)

# 诸论

* 1. **研究背景与意义**

近几十年来，随着计算机硬件和软件技术的不断发展，人们的娱乐方式有了巨大的变化，电子游戏就是其中之一。电子游戏在整个计算机产业的带动下不断地创新和发展。自从计算机发明，向各个领域发展，到成为我们现在每天工作和生活必不可少的一部分的这个过程中，电子游戏也逐步渗入到我们每个人的娱乐活动当中。而计算机已经普及的今天，对于可以用计算机进行程序编辑的人来说，开发属于自己的游戏，已经不再是梦想。游戏的种类繁多，其中益智类游戏相较于其他类型的游戏更有益身心健康，现如今，象棋游戏比比皆是，但由不同玩法和规则建立的象棋游戏并不多，如果象棋游戏有新的创新和玩法，那么将会给益智类游戏贡献出新的力量，也将会让玩家改变新的思维模式来面对这个游戏。

* 1. **决战象棋设计研究方法**

本系统使用C#语言和Unity 3D游戏引擎开发。一方面，C#的事件处理机制对游戏开发非常友好。另一方面，Unity 3D不仅支持C#语言开发，而且提供的API和强大的UGUI可视化编辑系统使游戏开发更加方便，因此，整体游戏功能会得到更好的完善。

* 1. **本文的主要工作**

本文的主要工作是将战斗属性结合到象棋中，建立一个全新的象棋计算机系统，研究工作主要集中在以下几个方面: 第一是资源重用，对于游戏来说，资源的优化既减轻硬件负担，也有利于提高游戏体验的流畅度。第二是行走规则，各兵种行走规则各异，都需要相应的算法实现。第一是战斗属性和计算，结合战斗属性，赋予棋子新的生命力，通过战斗计算决出胜负。第四是计时，计时跟回合切换息息相关，而本系统的回合切换因加入战斗属性而变得更为复杂，因此计时也会作出相应的改变。最后是悔棋和复盘，悔棋和复盘都需要记录棋谱，而且也因加入战斗属性而变得不一样。

* 1. **本文的结构**

第一章阐述了选题背景，课题的研究方式，课题的主要工作和文章的章节安排。第二章是系统的分析和总体设计。第三章是给出了实验环境和程序实现。第四章是全文的总结及展望。

# 系统的分析和设计

* 1. **需求分析**
     1. 界面需求

开始界面，可供操作的只有一个开始按钮，点击开始按钮后进入第二个界面，即加载界面。



图2.1 开始界面

加载界面有进度条，用于显示第三个界面加载进度，进度条满后，自动进入游戏主界面。

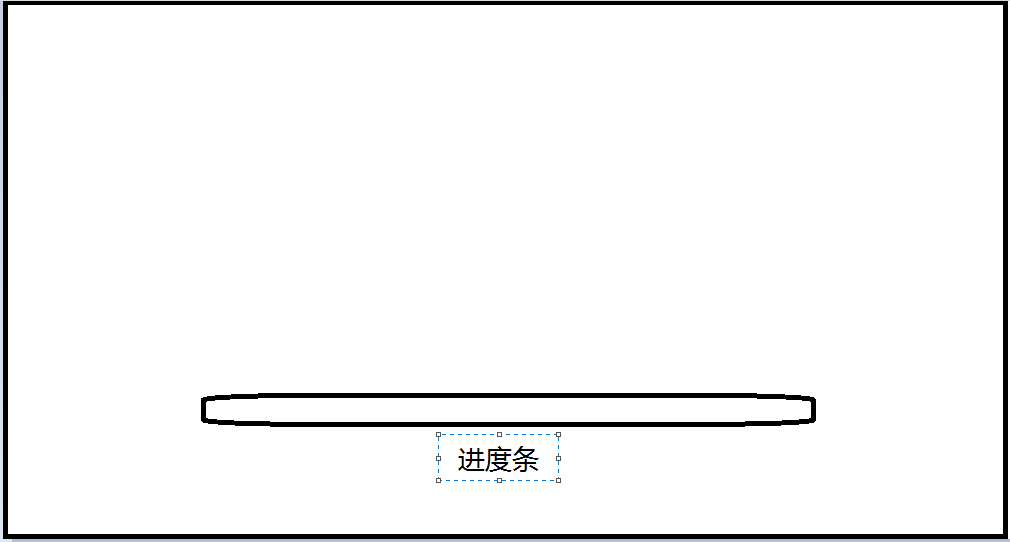


图2.2 加载界面

游戏主界面分为游戏进行时界面和复盘时界面两种模式。

游戏进行时界面有：局时，用于本局总剩余用时；步时，用于当前这一步限定用时；棋子，点击己方棋子时，在棋盘上会显示出可走点，根据行走规则，第二次点击有效可走点时会进行走步。第二次点击原棋子会取消选择。第二次点击其他己方棋子会更换选择；棋子属性面板，点击己方棋子时，右侧会显示本棋子具体属性，包括生命，攻击，防御，战力；其中按钮有：悔棋按钮、求和按钮、认输按钮、返回按钮、加生命属性按钮、加攻击属性按钮和加防御属性按钮。



图2.3 游戏主界面

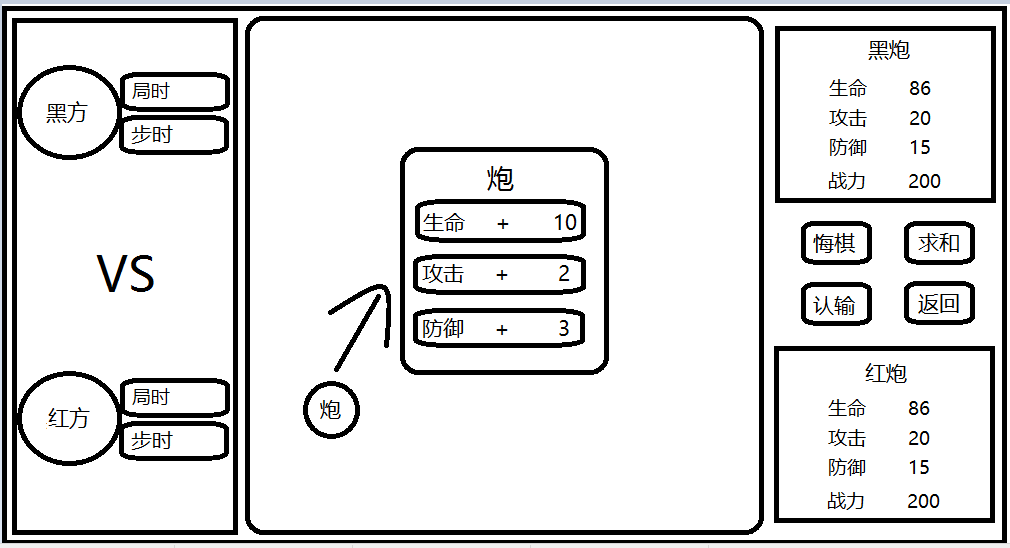


图2.4 游戏主界面的加属性情况

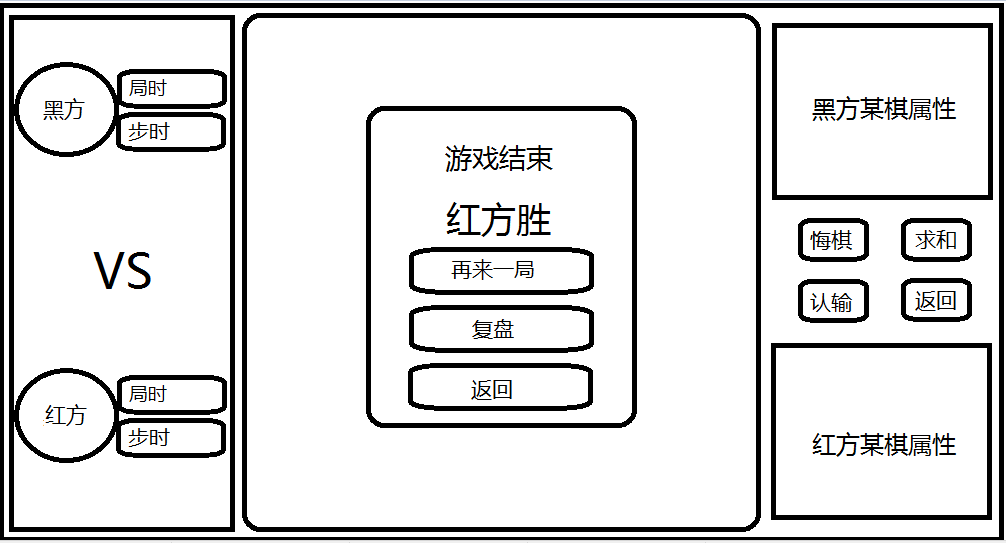


图2.5 游戏主界面的结束界面

复盘时界面有：某棋损失生命显示文本，用于显示吃子时胜方丢失血量；某棋加属性显示文本，用于显示每回合双方加属性情况；总步数文本，用于显示总步数；其中按钮有：返回按钮、开局按钮、终局按钮、上一步按钮和下一步按钮。



图2.6 游戏主界面复盘模式

* + 1. 非功能性需求

决战象棋游戏有具体如下规则:

1. 每人每回合操作有两步骤，分别是走步与加属性。
2. 各棋子走法遵循普通中国象棋的各棋子行走规则。
3. 每人走完步后，除自己的将帅之外，可以给其他棋子加一定属性。
4. 当完成走步和加属性操作后，计时才会停止，对方才开始计时。
5. 吃子行为属于战斗行为，胜败之分参考战斗计算方式。
6. 将帅无视战斗计算方式，可以无损吃子。
7. 棋局胜负判定遵循普通中国象棋规则。

决战象棋游戏的开发需要有状态要求，具体分为四大部分。第一个是游戏整体：未开始、进行中、暂停、复盘和结束共5个状态。第二个是游戏进行时：轮到红方走棋、轮到黑方走棋、轮到红方加属性和轮到黑方加属性共4个状态。第三个是棋子交互状态：未被选择、被选中和正在移动共3个状态。第四个是棋子所处棋局状态：安然无恙、攻击状态（将军）、被攻击（被将军）、被称为目标，敌人可吃、无路可走和阵亡共7个状态。

* 1. **开发所需条件和相关技术**

操作系统为Window10系统，游戏引擎为Unity3D，开发环境为Visual Studio 2017，开发语言为C#和Json。测试环境为Window10 PC端和Android移动端。

* 1. **总体设计**

本系统设计遵循MVC架构。MVC架构分为三层，分别是：视图层（View）、控制层（Control）和数据层（Model）。

视图层（View）：这层为展现给玩家的逻辑层，包括UI和场景，View中的任何逻辑不对Model层的数据进行修改，它对Model层拥有只读权限。

控制层（Control）：对Model层拥有读写权限。一般功能包括：（1）判断玩家操作是否符合条件，若符合执行指令。（2）根据逻辑对Model层中的数据进行修改。（3）回调View层中的方法通知操作完成或者失败。

数据层（Model）：主要是存放各模块的相关数据同时提供修改数据的方法。

根据本系统的开发需求，作出了以下整体的脚本规划：

视图层有3个UI场景，需要3个对应的UI脚本。另外，场景的异步加载状态信息监听需要1个脚本。控制层包括对整体游戏的控制，物体的生成，资源的回收和时间的限制，需要4个对应的控制管理类脚本。数据层负责存取数据的脚本和提供修改数据的方法，需要2个脚本。动态和需要获取信息的游戏物体主要有棋子和棋盘可走点。棋子可以抽象出共有的属性和行为，需要1个父类脚本。接着，共有车、马、炮、士、象、兵、帅七个兵种，都拥有不同行走规则，需要7个子类脚本。棋盘可走点物体需要1个脚本来控制状态。对游戏各模块的状态定义和事件声明，需要1个脚本。对棋子的属性，要有一个属性载体和属性配置信息相关处理，需要2个脚本。各棋子初始属性配置表，这个为txt文件即可，语法格式为Json。

* 1. **数据结构**

数据结构是一个程序的骨架，选择一种好的数据结构可以使程序的运行效率大大提高。本系统需要存储的数据主要是棋子，棋子属性和棋盘点。这三种数据类型又有紧密的联系，具体表现为：若已知棋子物体本身，则可以访问到其属性，也可以访问到其所处的位置；若已知棋盘坐标点，则可以得知点上是否存在棋子，是什么棋子。因此，若使用数组来存储以上的数据类型就很难取得上述关联信息，而C#有一种称为字典（Dictionary）的键值对数据结构，能很好的处理棋子、棋子属性、棋盘点的关系。例如，棋子和棋盘点之间的关系可通过字典来存储，因为棋子在任何时刻只会存在于某一棋盘点上，相反同理，在任何时刻，某一棋盘点上至多存在一颗棋子，因此两者可以建立双向键值对关系，这样可以达到知其一可得全部信息的目的。类似的，棋子和棋子属性之间的关系也可通过字典来存储，因为不同的棋子可能具有相同的棋子属性信息，所以两者只可建立单向键值对关系，即棋子为键，棋子属性为值。

* 1. **资源重用**

游戏性能的优劣取决于资源的消耗量，为减轻CPU运算和内存占用的负担，本系统资源重用主要应用在三个方面：

1.棋子阵亡时，回收到池而不是直接销毁物体对象。

2.再来一局时，从池里面提取所有棋子而不是重新生成棋子。

3.复盘和悔棋时，充池里面提取相应棋子而不是重新生成棋子。

* 1. **战斗属性处理**

每颗棋子都有3种属性，分别为生命，攻击，防御。不同兵种初始属性都不同，具体配置通过Json文本读取。战斗力是对这3个属性的总体评价。

战斗力计算方式：

Combat为战斗力，Hp为生命，Attack 为攻击，Defence为防御。

Combat = Hp\*0.4 + Defence\*10 + Attack\*10

A,B双方决斗，生死计算如下：

假设：a\_Hp = 100, a\_Attack = 4, a\_Defence = 2，b\_Hp = 120, b\_Attack = 6, b\_Defence = 1。按最小回合数为准，生命最先低于0的死亡，胜出方扣除最小回合数\*(对方伤害-自 身防御)的生命值

a对b造成的伤害a\_demage = a\_Attack-b\_Defence，若相减是负数，那么a\_demage=1。

b对a造成的伤害b\_demage =b\_Attack-a\_Defence，若相减是负数，那么b\_demage=1。

a攻击：b\_Hp/a\_demage = 40 （回合)

b攻击：a\_Hp/b\_demage = 25 （回合)

因此，b最先把a打死。

若a为进攻方，b为防守方，那么b用了25回合，a用了25回合，a阵亡。

minTimes = 25

b剩余血量：hp = b\_Hp - (a\_Attack-b\_Defence) \* minTimes = 120 - 75 = 45

若b为进攻方，a为防守方，那么b用了25回合，a用了24回合，a阵亡。

minTimes = 24

b剩余血量：hp = b\_Hp - (a\_Attack-b\_Defence) \* minTimes = 120 - 72 = 48

# 系统实现

* 1. **主体框架实现**

本系统的主体框架实现主要有资源重用的实现、数据层的实现和总控制器的实现，有了主体框架，后续的开发才能更加方便。

* + 1. 资源重用实现

棋子在一开始被生成时即加入到工作区，棋子阵亡时会被回收到回收区，当棋子再次出现在棋盘上时，会从回收期区提取出棋子并加入到工作区。工作区和回收区都使用GameObject类型的列表来表示。当棋子在这两个区发生变动时，有提供通知各模块的事件声明。

public class PoolManager : MonoBehaviour

{

public static event PushEventHandler PushEvent; //刚加入工作区事件

public static event TakeEventHandler TakeEvent; //从回收区提取棋子事件

public static event RestoreEventHandler RestoreEvent; //某个棋子回收事件

public static List<GameObject> work\_List; //工作区列表

public static List<GameObject> restore\_List; //回收区列表

public static Transform restoreChesses; //回收区父物体

public static Transform workChesses; //工作区父物体

void Awake()

{

restoreChesses = GameObject.Find("RestoreChesses").transform;

workChesses = GameObject.Find("WorkChesses").transform;

work\_List = new List<GameObject>();

restore\_List = new List<GameObject>();

}

/// 将创建的棋子加入工作区

public static void Push(GameObject chess)

{

work\_List.Add(chess);

SetParentToWork(chess);

PushEvent(chess);

}

/// 提取棋子

public static void Take(GameObject chess)

{

work\_List.Add(chess);

if (restore\_List.Contains(chess))

restore\_List.Remove(chess);

SetParentToWork(chess);

chess.SetActive(true);

TakeEvent(chess);//把取消订阅的事件全订阅回来

}

/// 回收棋子

public static void Restore(GameObject chess)

{

restore\_List.Add(chess);

if (work\_List.Contains(chess))

work\_List.Remove(chess);

SetParentToRestore(chess);

chess.SetActive(false);

RestoreEvent(chess);

}

public static void SetParentToWork(GameObject chess)

{

chess.transform.SetParent(workChesses, true);

}

public static void SetParentToRestore(GameObject chess)

{

chess.transform.SetParent(restoreChesses, true);

}

public void ClearPool()

{

if (work\_List != null)

work\_List.Clear();

if (restore\_List != null)

restore\_List.Clear();

}

public void OnDestroy()

{

ClearPool();

}

}

* + 1. 数据层实现

游戏中需要存取的数据分别为：场景中棋盘可走点物体与二维坐标的映射关系；当前棋局棋子与二维坐标的映射关系；当前棋局二维坐标与棋子的映射关系；棋局图谱即棋子与二维坐标的映射字典列表；棋子属性图谱即棋子与其属性的映射字典列表；棋子与其阵亡时对应步数的映射关系。还要提供相对应的修改数据和获取数据的方法。

public class GameCache

{

private static Dictionary<Vector3, Vector2> coords;

private static Dictionary<GameObject, Vector2> chess2Vector;

private static Dictionary<Vector2, GameObject> vector2Chess;

public static List<Dictionary<GameObject, Vector2>> maps;

public static List<Dictionary<GameObject, string>> attrMaps;

private static Dictionary<GameObject, int> loserStepDic;

public static void SetCoords(GameObject[,] cells)

{

if (coords == null)

{

coords = new Dictionary<Vector3, Vector2>();

for (int x = 0; x <= 8; x++)

{

for (int y = 0; y <= 9; y++)

{ //强制为整形，消除微弱差别的影响

int a = (int)cells[x, y].transform.position.x;

int b = (int)cells[x, y].transform.position.y;

int c = (int)cells[x, y].transform.position.z;

coords.Add(new Vector3(a, b, c), new Vector2(x, y));

}

}

}

}

public static void SetChessVectorDic(List<GameObject> chessList)

{

chess2Vector = new Dictionary<GameObject, Vector2>();

vector2Chess = new Dictionary<Vector2, GameObject>();

for (int i = 0; i < chessList.Count; i++)

{

Vector3 pos = chessList[i].transform.position;

Vector3 v3 = new Vector3((int)pos.x, (int)pos.y, (int)pos.z);

chess2Vector.Add(chessList[i], coords[v3]);

vector2Chess.Add(coords[v3], chessList[i]);

}

}

public static void SetLoserStepDic(GameObject loser, int step)

{

//悔棋和复盘将会提到

}

public static void ClearChessVectorDic()

{

if(Chess2Vector != null || Vector2Chess != null)

{

chess2Vector.Clear();

vector2Chess.Clear();

}

}

/// 更新棋局信息

public static void UpdateChessData()

{

ClearChessVectorDic();

SetChessVectorDic(PoolManager.work\_List);

}

/// 添加棋谱

public static void SetMaps()

{

//悔棋和复盘将会提到

}

/// 添加每回合全部棋子对应属性信息图谱

public static void SetAttrMaps()

{

//悔棋和复盘将会提到

}

public static void ClearCache()

{

if (maps != null)

maps.Clear();

if (attrMaps != null)

attrMaps.Clear();

if (loserStepDic != null)

loserStepDic.Clear();

ClearChessVectorDic();

}

}

* + 1. 游戏控制器的实现

游戏控制器GameController是控制整个游戏动向的脚本，它负责记录整个游戏状态并按需发送改变游戏状态的相关指令，具体表现为：当场景切换时，整体游戏状态设置为未开始，清空数据层缓存，步数归零，订阅和取消订阅相关事件等。当玩家游戏操作完成时，负责判断是否发生战斗和改变回合状态，再发送指令将当前棋局信息写入数据层缓存。复盘模式时，负责重置游戏数据和相关状态。游戏控制器在整个系统中有且只有一个，使用单例模式来设计。

public class GameController : MonoBehaviour

{

void Awake()

{

if (instance == null)

instance = this;

else

{

Destroy(gameObject);

}

SceneManager.sceneLoaded += OnSceneLoaded; //加载场景事件监听

if (hadLoad == false)

{

hadLoad = true;

DontDestroyOnLoad(gameObject);//只执行最开始的那一遍

}

}

private void OnSceneLoaded(Scene scene, LoadSceneMode arg1)

{

IsBattle = false;

step = 0;

gameStatus = GameStatus.NotBegin;

playing = Playing.None;

GameCache.ClearCache();

if (scene.name.Equals("scene3(Main)")) //若主场景加载完毕并切换到主场景

{

Debug.Log("主场景已加载完毕并切换到主场景");

Scene3\_UI.AddAttrCompleteEvent += UpdateBout;

}

else

{

BaseChess.SetAttackerEvent -= SetAttacker;

BaseChess.SetDefenderEvent -= SetDefender;

Scene3\_UI.AddAttrCompleteEvent -= UpdateBout;

}

}

public static void SetAttacker(GameObject chess){attacker = chess;}

public static void SetDefender(GameObject chess){defender = chess;}

public void UpdateGame()

{

if (gameStatus == GameStatus.Going)

{

if (IsBattle == true)

{

IsBattle = false;

GameObject loser = GameUtil.Battle(attacker, defender);

GameCache.SetLoserStepDic(loser, step);

KilledEvent(loser);

}

UpdateBout();

}

}

/// 更新回合

public void UpdateBout()

{

if (playing == Playing.None)

{

UpdateGameData();

playing = Playing.OnRed;

}

else if (playing == Playing.OnRed)

playing = Playing.RedAdding;

else if (playing == Playing.RedAdding)

{

UpdateGameData();

playing = Playing.OnBlack;

}

else if (playing == Playing.OnBlack)

playing = Playing.BlackAdding;

else if (playing == Playing.BlackAdding)

{

UpdateGameData();

playing = Playing.OnRed;

}

}

public void UpdateGameData()

{

GameCache.UpdateChessData();

GameCache.SetMaps();

GameCache.SetAttrMaps();

UpdateGameDataCompleteEvent();

step++;

}

public static void BeginGame(){ }

public static void ResetGame(){ }

public static void ReplayModeGame(){ }

}

* 1. **界面实现**

场景一UI界面摆放如图：



图3.1 开始界面

场景一界面比较简单，只有一个开始游戏按钮，点击则进入场景二。

场景二UI界面摆放如图：

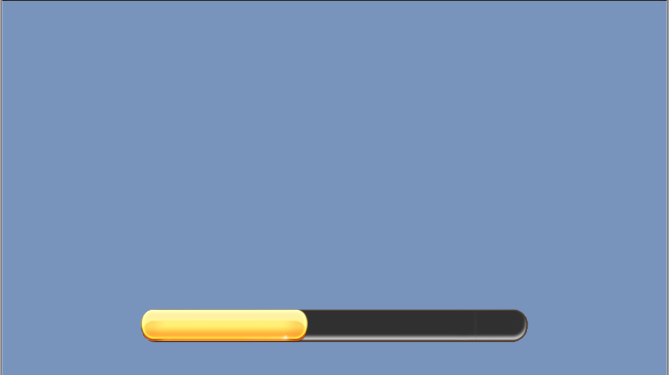


图3.2 加载界面

场景三是加载界面，由场景一跳转进入，等待场景三物体加载完毕后自动跳转到场景三界面。这个过程需要异步加载。进度条显示场景三的加载进度。

public class LoadingByAsync : MonoBehaviour

{

AsyncOperation async; //异步对象

public static LoadingByAsync instance;

public Slider loadingSlider;

void Awake()

{

if (instance == null)

instance = this;

loadingSlider=GameObject.Find("LoadingSlider").GetComponent<Slider>();

}

void Start()

{

StartCoroutine(DelayLoading());

}

void Update()

{

if (async == null)

return;

int toProcess;

//async.progress的取值范围在0.1-1之间,但是它不会等于1,加载完也就等于0.9

if (async.progress < 0.9f)

toProcess = (int)async.progress \* 100;

else

toProcess = 100;

if (loadingSlider.value < toProcess)

loadingSlider.value = toProcess;

if (toProcess == 100)

async.allowSceneActivation = true;

}

public IEnumerator DelayLoading()

{

yield return new WaitForSeconds(0.1f);

yield return Load\_Scene("scene3(Main)");

}

//注意这里返回值一定是 IEnumerator

public IEnumerator Load\_Scene(string sceneName)

{

//异步读取场景。

async = SceneManager.LoadSceneAsync(sceneName);

async.allowSceneActivation = false;

yield return async;

}

}

场景3游戏进行时界面如图：

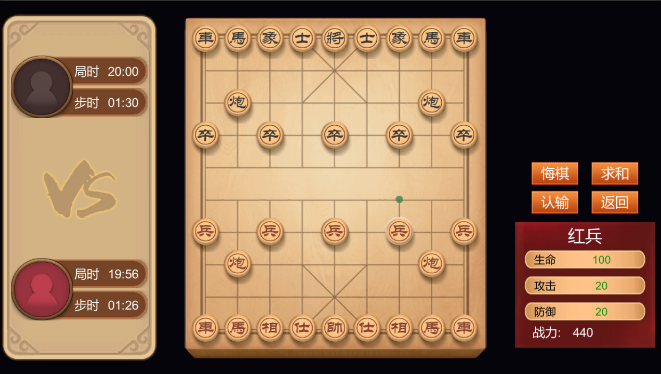


图3.3 游戏主界面1



图3.4 游戏主界面2

场景3复盘时界面如图：

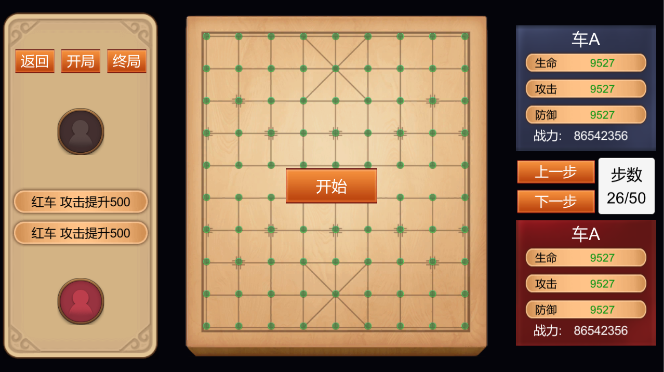


图3.5 游戏主界面复盘模式

场景3游戏进行时加属性如图:

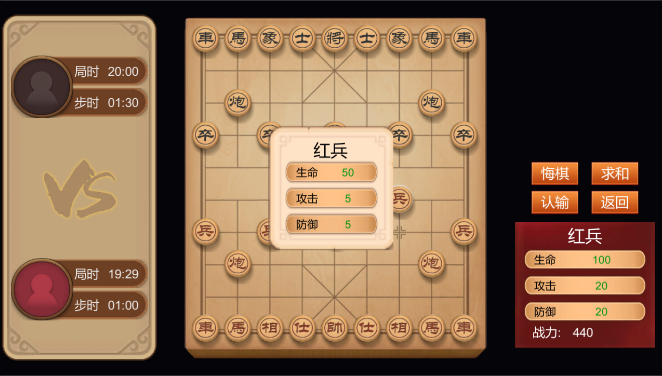


图3.6 游戏主界面加属性情况

场景3的UI元素比较多，需要分情况处理。具体如下：

按游戏模式划分：

1.游戏进行时，左侧显示双方局时和步时，右侧中部显示四个供玩家操作的按钮，分别是悔棋、求和、认输和返回按钮。右侧相应位置显示当前行走方选中棋子的属性面板。

2.游戏复盘时，左侧显示双方当前步数棋子属性发生修改的情况。左侧上方显示三个供玩家操作的按钮，分别是返回、开局和终局按钮。右侧中部显示步数情况和两个供玩家操作的按钮，分别是上一步和下一步按钮。

按游戏进展状态划分：

1.当一方走步完成后，再次点击棋子，中间显示出加属性面板，其中有三个供玩家操作的按钮，分别是加生命，加攻击和加防御按钮。

2.当游戏结束时，中间显示出游戏结束面板，其中有三个供玩家操作的按钮，分别是再来一局，复盘和返回按钮。游戏结束时关闭其余面板并且禁止其他操作。

public class Scene3\_UI : MonoBehaviour

{

/\*这里属性声明和初始化省略\*/

void Update()

{

if (GameController.gameStatus == GameStatus.Going){ }

}

public void SetMode(bool isGameMode){ }

public void OnBeginClick(){ }

public static void ResetChessBoardPoints(){ }

public void OnChoose(GameObject chess){ }

public void UpdateAttrPanel(){ }

public void UpdateAttrPanel(GameObject chess){ }

public void UpdateReplayModeText()

{

stepsLabel.text = GameController.step+"/"+(GameCache.maps.Count - 1);

}

public void OnHpClick(){ }

public void OnAttackClick(){ }

public void OnDefenceClick(){ }

public void HideAddAttrPanel(){ }

public void HideAttrPanel(){ }

// 设置属性面板

public void SetAttrTexts(AttrBox attrBox, GameObject chess){ }

public void ShowEndPanel(string loserStr){ }

public void HideEndPanel(){ }

public void OnPlayAgainClick(){ }

public void OnReplayModeClick(){ }

public void OnBackClick(){ }

public void OnUndoClick()

{

if (GameCache.maps.Count >= 3)

{

//回合状态回退

if (GameController.playing == Playing.OnRed || GameController.playing == Playing.RedAdding)

GameController.playing = Playing.OnRed;

else

GameController.playing = Playing.OnBlack;

//棋谱回退 并移除回退前两步信息

GameUtil.SetChessBoardByMaps(GameCache.maps.Count - 1 - 2);

GameCache.maps.RemoveRange(GameCache.maps.Count - 2, 2);

GameCache.attrMaps.RemoveRange(GameCache.attrMaps.Count-2,2);

GameCache.UpdateChessData(); //更新映射关系，但不计入图谱

UndoEvent();

}

}

public void OnGiveUpClick(){ }

public void OnSuePeaceClick(){ }

public void OnPreStepClick()

{

if (GameController.gameStatus == GameStatus.Replay)

{

GameController.step--;

if (GameController.step < 0)

{

GameController.step = 0;

return;

}

GameUtil.SetChessBoardByMaps(GameController.step);

UpdateReplayModeText();

}

}

public void OnNextStepClick()

{

if (GameController.gameStatus == GameStatus.Replay)

{

GameController.step++;

if (GameController.step > GameCache.maps.Count - 1)

{

GameController.step = GameCache.maps.Count - 1;

return;

}

GameUtil.SetChessBoardByMaps(GameController.step);

UpdateReplayModeText();

}

}

public void OnBeginStepClick()

{

if (GameController.gameStatus == GameStatus.Replay)

{

GameController.step = 0;

GameUtil.SetChessBoardByMaps(0);

UpdateReplayModeText();

}

}

public void OnEndStepClick()

{

if (GameController.gameStatus == GameStatus.Replay)

{

GameController.step = GameCache.maps.Count - 1;

GameUtil.SetChessBoardByMaps(GameCache.maps.Count - 1);

UpdateReplayModeText();

}

}

* 1. **棋盘表示**

在场景3中，中部显示棋盘信息。棋盘外观用一张背景图表示。棋盘内共有10\*9个行走点，需要制作90个行走点物体。因为游戏场景中行走点物体的坐标是以世界坐标计算的，所以还需要将行走点物体对应回以左下角为原点的二维坐标。为了计算方便，这90个行走点物体需要按一定的顺序摆放。先按行分组，从下到上共10行即y坐标区间从下到上[0,9]。再按列摆放，每一行共9个行走点即x坐标区间从左到右[0,8]。

GridsTrans = GameObject.Find("Grids").transform;

cells = new GameObject[9, 10];

for (int y = 0; y <= 9; y++)

{

Transform row = GridsTrans.FindChild("row" + y);

for (int x = 0; x <= 8; x++)

{

cells[x, y] = row.FindChild("cell" + x).gameObject;

cells[x, y].GetComponent<Image>().enabled = false;

}

}

GameCache.SetCoords(cells);

//将场景找到的cell作为参数，处理写入映射缓存

public static void SetCoords(GameObject[,] cells)

{

if (coords == null)

{

coords = new Dictionary<Vector3, Vector2>();

for (int x = 0; x <= 8; x++)

{ //强制为整形，消除微弱差别的影响

for (int y = 0; y <= 9; y++)

{

int a = (int)cells[x, y].transform.position.x;

int b = (int)cells[x, y].transform.position.y;

int c = (int)cells[x, y].transform.position.z;

coords.Add(new Vector3(a, b, c), new Vector2(x, y));

}

}

}

}

棋盘上还需要有棋子显示，需要制作所有棋子的预制体，然后在指定位置生成出来，生成时将棋子加入到池工作区进行管理，以便资源重用，下面以生成红车为例。

public Transform Create(GameObject prefab, int x, int y)

{

GameObject go = Instantiate(prefab);

PoolManager.Push(go); //放入池工作区进行管理

go.transform.position = Scene3\_UI.cells[x, y].transform.position;

go.transform.localScale = Vector3.one;

bornMap.Add(go, new Vector2((int)x, (int)y));

return go.transform;

}

public void InitChessBoard()

{

b\_Ju1 = Create(black\_Ju, 0, 9);

}

可走点需要响应玩家的点击事件，编写脚本为PointCell.cs 。点击棋子，当棋子有可走的点时，可走点显示为绿色。当点击可走点时，发送事件通知棋子移动到此点上。

public class PointCell : MonoBehaviour

{

[HideInInspector]

public Vector2 point;

[HideInInspector]

public Transform parent;

public static event MoveEventHandler PointCellClickEvent;

void Awake(){

parent = transform.parent;

point = GameUtil.Str2Vector(parent.name, gameObject.name);

}

public void OnCellClick(){

PointCellClickEvent(point);

}

}

* 1. **棋子实现**
     1. 共有属性和行为

首先，要抽象出棋子的共有属性和行为方法，脚本为BaseChess.cs。

共有属性有：棋子的交互状态、棋子的形势状态、棋子中文名称、棋子变量名称（供属性处理使用）和战斗属性，其中战斗属性则封装成一个类，依附在棋子身上，方便移除和批量写入属性的处理。

共有行为和方法有：被点击、被选中、取消选中、提示为被攻击目标、移动、可移动点集合、阵亡、重置自身状态的方法以及订阅和取消订阅相关事件的方法。

棋子被点击时，首先判断当前回合状态，如果是轮到己方操作，那么被点击会认为是被选中，再次点击则取消选中。如果是轮到敌方操作，那么再判断如果棋子自身标记为被攻击目标，那么将会通知敌方棋子移动到当前位置并进行战斗。

public virtual void ChesseClicked()

{

if (GameController.playing == Playing.OnRed)

{

if (gameObject.tag == "Red")

{

if (chessReciprocalState==ChessReciprocalState.unChoosed)

{

ChooseEvent(gameObject);

BeChoosed(false);

}

else

{

CancelChoose();

Scene3\_UI.ResetChessBoardPoints();

}

}

else

{

if (chessSituationState == ChessSituationState.BeTaget)

{//若被选中且被标记为红点，就把自身物体派发出去

GameController.IsBattle = true;

SetDefenderEvent(gameObject); //将自身防守方派发出去

MoveEvent(GameCache.Chess2Vector[gameObject]);

}

else return;

}

}

else if (GameController.playing == Playing.OnBlack)

{

if (gameObject.tag == "Black")

{

if(chessReciprocalState == ChessReciprocalState.unChoosed)

{

ChooseEvent(gameObject);

BeChoosed(false);

}

else

{

CancelChoose();

Scene3\_UI.ResetChessBoardPoints();

}

}

else

{

if (chessSituationState == ChessSituationState.BeTaget)

{//若被选中且被标记为红点，就把自身物体派发出去

GameController.IsBattle = true;

SetDefenderEvent(gameObject); //将自身防守方派发出去

MoveEvent(GameCache.Chess2Vector[gameObject]);

}

else return;

}

}

else if (GameController.playing == Playing.RedAdding)

{

if (gameObject.tag == "Red")

{

ChooseEvent(gameObject);

BeChoosed(true);

}

else return;

}

else if (GameController.playing == Playing.BlackAdding)

{

if (gameObject.tag == "Black")

{

ChooseEvent(gameObject);

BeChoosed(true);

}

else return;

}

}

/// 被选中

public void BeChoosed(bool isAdding)

{

transform.FindChild("baibian").gameObject.SetActive(true);

if (!isAdding)

{

Scene3\_UI.ResetChessBoardPoints();

//可以提示出该棋子能移动的所有位置

Vector2[] canMovePoints = CanMovePoints(GameCache.Chess2Vector, GameCache.Vector2Chess).ToArray();

for (int i = 0; i < canMovePoints.Length; i++)

{

int x = (int)canMovePoints[i].x; int y = (int)canMovePoints[i].y;

Scene3\_UI.cells[x, y].GetComponent<Image>().enabled = true;

//若可移动点上存在其他棋子，那肯定就是敌方棋子了，提示可以击杀之

if (GameCache.Vector2Chess.ContainsKey(canMovePoints[i]))

{

GameCache.Vector2Chess[canMovePoints[i]].transform.FindChild("redpoint").gameObject.SetActive(true);

TipsKillEvent(canMovePoints[i]);

}

}

chessReciprocalState = ChessReciprocalState.beChoosed;

}

}

public void CancelChoose(GameObject chess)

{

if (gameObject != chess)

Reset(); //将被选中时的所有变化还原

}

当对方棋子被选中时，棋盘上显示出对方棋子可走的点。如果可走点上存在敌方棋子，那么敌方棋子会被标记为攻击目标。这个方法需要监听棋子被选中时的事件。

public void TipsBeTarget(Vector2 point)

{

if (GameUtil.CompareVector2(GameCache.Chess2Vector[gameObject], point))

chessSituationState = ChessSituationState.BeTaget;

}

棋子移动点集合在子类中具体实现，在基类中只作抽象声明，供移动方法调用。

/// 根据棋局信息，该棋子能移动的所有位置,返回的是平面二维坐标，如(0,1)

public abstract List<Vector2> CanMovePoints(Dictionary<GameObject, Vector2> chess2Vector, Dictionary<Vector2, GameObject> vector2Chess);

棋子移动方法，根据棋子状态和移动点集合过滤其他非法操作，当符合规则并可以移动时判断是否属于移动并攻击的行为，如果是则移动完成后回调战斗比较方法，否则只算是普通移动行为。

public void Move(Vector2 point)

{

if (chessReciprocalState == ChessReciprocalState.beChoosed)

{

Vector2[] canMovePoints = CanMovePoints(GameCache.Chess2Vector, GameCache.Vector2Chess).ToArray();

if (canMovePoints.Length > 0)

{

for (int i = 0; i < canMovePoints.Length; i++)

{

if(GameUtil.CompareVector2(point, canMovePoints[i]) == true)

{

if(GameController.IsBattle == true) //派发事件设置攻击者

SetAttackerEvent(gameObject);

int x = (int)point.x; int y = (int)point.y;

Vector3 target = Scene3\_UI.cells[x, y].transform.position;

iTween.MoveTo(gameObject, iTween.Hash("time", 0.1f, "position", target,

"easetype", iTween.EaseType.linear,

"onstart", "OnMoveStart",

"onstarttarget", gameObject,

"oncomplete", "OnMoveComplete",

"oncompletetarget", gameObject));

break;

}

else if (chessReciprocalState != ChessReciprocalState.moving)

{

CancelChoose();

Scene3\_UI.ResetChessBoardPoints();

}

else

{

CancelChoose();

Scene3\_UI.ResetChessBoardPoints();

}

}

}

else

CancelChoose();

}

}

当棋子战斗失败时，棋子阵亡，游戏控制器通知回收此棋子。

public virtual void Killed(GameObject chess)

{

if (chess == gameObject) //被杀 回收

PoolManager.Restore(gameObject);

}

* + 1. 各兵种行走规则

棋子行走规则的具体表现也就是可移动点的集合。各兵种的行走规则不同，意味着他们的可移动点分布规律也不同。所以，各兵种棋子脚本需要具体实现可移动点集合方法。

车：车在象棋中威力最大，无论横线、竖线均可行走，只要无子阻拦，步数不受限制。俗称“车行直路”。因此，一车可以控制十七个点，故有“一车十子寒”之称。车行走点的统计分上、下、左、右四个方向进行。以向右为例，根据车目前所处位置求出车右边的棋盘点个数，根据这些棋盘点循环遍历，查找存在棋子的棋盘点。若棋盘点不存在棋子，则说明此棋盘点是车的可行走点。否则，终止后续遍历，然后再判断棋子是否是己方棋子，若是则此棋盘点不可走，否则此棋盘点可行走并且会认定为战斗点。

public class Chess\_Ju : BaseChess

{

public override List<Vector2> CanMovePoints(Dictionary<GameObject, Vector2> chess2Vector, Dictionary<Vector2, GameObject> vector2Chess)

{

Vector2 currentPos = chess2Vector[gameObject];

List<Vector2> canMovePoints = new List<Vector2>();

for (int i = (int)currentPos.x - 1; i >= 0; i--) //向左检索

{

Vector2 value = new Vector2(i, currentPos.y);

bool findOtherChess = false;

JudgeMovePoint(value, ref findOtherChess, canMovePoints, vector2Chess);

if (findOtherChess) break;

}

for (int i = (int)currentPos.x + 1; i <= 8; i++) //向右检索

{

Vector2 value = new Vector2(i, currentPos.y);

bool findOtherChess = false;

JudgeMovePoint(value, ref findOtherChess, canMovePoints, vector2Chess);

if (findOtherChess) break;

}

for (int i = (int)currentPos.y + 1; i <= 9; i++) //向上检索

{

Vector2 value = new Vector2(currentPos.x, i);

bool findOtherChess = false;

JudgeMovePoint(value, ref findOtherChess, canMovePoints, vector2Chess);

if (findOtherChess) break;

}

for (int i = (int)currentPos.y - 1; i >= 0; i--) //向下检索

{

Vector2 value = new Vector2(currentPos.x, i);

bool findOtherChess = false;

JudgeMovePoint(value, ref findOtherChess, canMovePoints, vector2Chess);

if (findOtherChess) break;

}

return canMovePoints;

}

void JudgeMovePoint(Vector2 value, ref bool findOtherChess, List<Vector2> canMovePoints, Dictionary<Vector2, GameObject> vector2Chess)

{

if (vector2Chess.ContainsKey(value)) //若有其他棋子，那就停下来

{

GameObject otherChess = vector2Chess[value];

if(otherChess.tag != gameObject.tag) //判断是否为己方棋子

canMovePoints.Add(value);

findOtherChess = true;

}

else

canMovePoints.Add(value);

}

}

马：马走动的方法是一直一斜，即先横着或直着走一格，然后再斜着走一个对角线，俗称“马走日”。马一次可走的选择点可以达到四周的八个点，故有“八面威风”之说。如果在要去的方向有别的棋子挡住，马就无法走过去，俗称“蹩马腿”。马行走点的统计分八个方向进行，此八个方向可以比作时钟的1、2、4、5、7、8、10、11点钟方向。以一点钟方向为例，假设当前马坐标位置为(x, y)，那么需要判断的是马上方(x, y+1)处是否存在绊马脚的棋子。若存在则马不可走一点钟方向的位置，否则继续判断(x+1, y+2)是否存在棋子。若不存在则马可走一点钟方向的位置，否则继续判断是否为己方棋子。若是则马不可走一点钟方向的位置，否则一点钟方向的位置是马的可走点并标记为战斗点。

public class Chess\_Ma : BaseChess

{

public override List<Vector2> CanMovePoints(Dictionary<GameObject, Vector2> chess2Vector, Dictionary<Vector2, GameObject> vector2Chess)

{

Vector2 currentPos = chess2Vector[gameObject];

bool stopHourseRight = false; //在右边绊马脚

bool stopHourseLeft = false; //在左边绊马脚

bool stopHourseUp = false; //在上边绊马脚

bool stopHourseDown = false; //在下边绊马脚

List<Vector2> canMovePoints = new List<Vector2>();

if (vector2Chess.ContainsKey(new Vector2(currentPos.x+1, currentPos.y)))

stopHourseRight = true;

if (vector2Chess.ContainsKey(new Vector2(currentPos.x-1, currentPos.y)))

stopHourseLeft = true;

if (vector2Chess.ContainsKey(new Vector2(currentPos.x, currentPos.y+1)))

stopHourseUp = true;

if (vector2Chess.ContainsKey(new Vector2(currentPos.x, currentPos.y-1)))

stopHourseDown = true;

if (stopHourseRight == false)

{

//2点钟

Vector2 v2 = new Vector2(currentPos.x + 2, currentPos.y + 1);

JudgeMovePoint(v2, canMovePoints, vector2Chess);

//4点钟

Vector2 v4 = new Vector2(currentPos.x + 2, currentPos.y - 1);

JudgeMovePoint(v4, canMovePoints, vector2Chess);

}

if (stopHourseDown == false)

{

//5点钟

Vector2 v5 = new Vector2(currentPos.x + 1, currentPos.y - 2);

JudgeMovePoint(v5, canMovePoints, vector2Chess);

//7点钟

Vector2 v7 = new Vector2(currentPos.x - 1, currentPos.y - 2);

JudgeMovePoint(v7, canMovePoints, vector2Chess);

}

if (stopHourseLeft == false)

{

//8点钟

Vector2 v8 = new Vector2(currentPos.x - 2, currentPos.y - 1);

JudgeMovePoint(v8, canMovePoints, vector2Chess);

//10点钟

Vector2 v10 = new Vector2(currentPos.x - 2, currentPos.y + 1);

JudgeMovePoint(v10, canMovePoints, vector2Chess);

}

if (stopHourseUp == false)

{

//11点钟

Vector2 v11 = new Vector2(currentPos.x - 1, currentPos.y + 2);

JudgeMovePoint(v11, canMovePoints, vector2Chess);

//1点钟

Vector2 v1 = new Vector2(currentPos.x + 1, currentPos.y + 2);

JudgeMovePoint(v1, canMovePoints, vector2Chess);

}

return canMovePoints;

}

void JudgeMovePoint(Vector2 value, List<Vector2> canMovePoints, Dictionary<Vector2, GameObject> vector2Chess)

{

if (GameUtil.IsInChessBoard(value))

{

if (vector2Chess.ContainsKey(value)) //若有棋子

{

GameObject otherChess = vector2Chess[value];

if (otherChess.tag != gameObject.tag)

canMovePoints.Add(value);

}

else

canMovePoints.Add(value);

}

}

}

炮：炮在不吃子的时候，走动与车完全相同，但炮在吃子时，必须跳过一个棋子，我方的和敌方的都可以，俗称“炮打隔子”、“翻山”。炮的行走点分上、下、左、右四个方向进行统计。以向右为例，根据炮目前所处位置求出炮右边的棋盘点个数，根据这些棋盘点循环遍历，查找存在棋子的棋盘点。若棋盘点不存在棋子，则说明此棋盘点是炮的可行走点。否则，在找到第一个棋子时做一个标记，然后继续寻找第二个棋子，然后再判断棋子是否是己方棋子，若是则此棋盘点不可走，否则此棋盘点可行走并且会认定为战斗点。

public class Chess\_Pao : BaseChess

{

public override List<Vector2> CanMovePoints(Dictionary<GameObject, Vector2> chess2Vector, Dictionary<Vector2, GameObject> vector2Chess)

{

Vector2 currentPos = chess2Vector[gameObject];

List<Vector2> canMovePoints = new List<Vector2>();

bool findFirstLeftOtherChess = false;

bool findFirstRightOtherChess = false;

bool findFirstUpOtherChess = false;

bool findFirstDownOtherChess = false;

for (int i = (int)currentPos.x - 1; i >= 0; i--) //向左检索

{

Vector2 value = new Vector2(i, currentPos.y);

bool findSecondChess = false;

JudgeMovePoint(value, ref findFirstLeftOtherChess, ref findSecondChess, canMovePoints, vector2Chess);

if (findSecondChess) break;

}

for (int i = (int)currentPos.x + 1; i <= 8; i++) //向右检索

{

Vector2 value = new Vector2(i, currentPos.y);

bool findSecondChess = false;

JudgeMovePoint(value, ref findFirstRightOtherChess, ref findSecondChess, canMovePoints, vector2Chess);

if (findSecondChess) break;

}

for (int i = (int)currentPos.y + 1; i <= 9; i++) //向上检索

{

Vector2 value = new Vector2(currentPos.x, i);

bool findSecondChess = false;

JudgeMovePoint(value, ref findFirstUpOtherChess, ref findSecondChess, canMovePoints, vector2Chess);

if (findSecondChess) break;

}

for (int i = (int)currentPos.y - 1; i >= 0; i--) //向下检索

{

Vector2 value = new Vector2(currentPos.x, i);

bool findSecondChess = false;

JudgeMovePoint(value, ref findFirstDownOtherChess, ref findSecondChess, canMovePoints, vector2Chess);

if (findSecondChess) break;

}

return canMovePoints;

}

/// 炮专属判断是否可以走这个点

void JudgeMovePoint(Vector2 value, ref bool findFirstOtherChess, ref bool findSecondChess, List<Vector2> canMovePoints, Dictionary<Vector2, GameObject> vector2Chess)

{

if (findFirstOtherChess == false) //若还没找到第一个棋子，就让他继续找

{

if (vector2Chess.ContainsKey(value))

findFirstOtherChess = true;

else

canMovePoints.Add(value);

}

else//找到了第一个棋子后，就找第二个

{

if (vector2Chess.ContainsKey(value))//找到第二个且是敌方棋子

{

GameObject targetChess = vector2Chess[value];

if (targetChess.tag != gameObject.tag)

canMovePoints.Add(value);

findSecondChess = true;

}

}

}

}

士/仕：红方为“仕”，黑方为“士”。它也只能在九宫内走动。它的行棋路径只能是九宫内的斜线。士一次只能走一个斜格。士的行走点分左斜向上45度、左斜向下45度、右斜向上45度和右斜向下45度四个方向进行统计。以左斜向上45度方向为例，根据士目前所处位置(x,y)判断左斜向上45度(x+1, y+1)处是否在九宫格内，若不是则斜向上45度方向没有可走点。否则，再继续判断该点是否存在棋子。若不存在则该点为士的可行走点，否则继续判断该点处的棋子是否为己方棋子。若是则该点不可走，否则该点为士的可行走点并标记为战斗点。

public class Chess\_Shi : BaseChess

{

public override List<Vector2> CanMovePoints(Dictionary<GameObject, Vector2> chess2Vector, Dictionary<Vector2, GameObject> vector2Chess)

{

Vector2 currentPos = chess2Vector[gameObject];

List<Vector2> canMovePoints = new List<Vector2>();

if (gameObject.tag == "Red")

{

if (currentPos.x < 5 && currentPos.y < 2) //45°斜向上走

{

Vector2 value = new Vector2(currentPos.x + 1, currentPos.y + 1);

JudgeMovePoint(value, canMovePoints, vector2Chess);

}

if (currentPos.x < 5 && currentPos.y > 0) //-45°斜向下走

{

Vector2 value = new Vector2(currentPos.x + 1, currentPos.y - 1);

JudgeMovePoint(value, canMovePoints, vector2Chess);

}

if (currentPos.x > 3 && currentPos.y < 2) //135°斜向上走

{

Vector2 value = new Vector2(currentPos.x - 1, currentPos.y + 1);

JudgeMovePoint(value, canMovePoints, vector2Chess);

}

if (currentPos.x > 3 && currentPos.y > 0) //-135°斜向下走

{

Vector2 value = new Vector2(currentPos.x - 1, currentPos.y - 1);

JudgeMovePoint(value, canMovePoints, vector2Chess);

}

}

if (gameObject.tag == "Black")

{

if (currentPos.x < 5 && currentPos.y < 9) //45°斜向上走

{

Vector2 value = new Vector2(currentPos.x + 1, currentPos.y + 1);

JudgeMovePoint(value, canMovePoints, vector2Chess);

}

if (currentPos.x < 5 && currentPos.y > 7) //-45°斜向下走

{

Vector2 value = new Vector2(currentPos.x + 1, currentPos.y - 1);

JudgeMovePoint(value, canMovePoints, vector2Chess);

}

if (currentPos.x > 3 && currentPos.y < 9) //135°斜向上走

{

Vector2 value = new Vector2(currentPos.x - 1, currentPos.y + 1);

JudgeMovePoint(value, canMovePoints, vector2Chess);

}

if (currentPos.x > 3 && currentPos.y > 7) //-135°斜向下走

{

Vector2 value = new Vector2(currentPos.x - 1, currentPos.y - 1);

JudgeMovePoint(value, canMovePoints, vector2Chess);

}

}

return canMovePoints;

}

/// <summary>

/// 士专属判断是否可以走这个点

/// </summary>

/// <param name="value"></param>

void JudgeMovePoint(Vector2 value, List<Vector2> canMovePoints, Dictionary<Vector2, GameObject> vector2Chess)

{

//若网格存在，即在棋盘内

if (GameUtil.IsInChessBoard(value))

{

//若有棋子

if (vector2Chess.ContainsKey(value))

{

GameObject otherChess = vector2Chess[value];

if (otherChess.tag != gameObject.tag)

canMovePoints.Add(value);

}

else

canMovePoints.Add(value);

}

}

}

象/相：红方为“相”，黑方为“象”。它的走法是每次循对角线走两格，俗称“象飞田”。相（象）的活动范围限于“河界”以内的本方阵地，不能过河，且如果它走的“田”字中央有一个棋子，就不能走，俗称“塞象眼”。象的行走点统计方向与士的统计方向一致，但行走范围比较广阔。以左斜向上45度方向为例，根据象目前所处位置(x,y)判断左斜向上45度(x+1, y+1)处是否存在棋子，若是则斜向上45度方向没有可走点。否则，再继续判断(x+2, y+2)处是否超越河界(黑方河界y值为5，红方河界y值为4)。若是则该点不可走。否则，再继续判断该点是否存在棋子。若不存在则该点为象的可行走点，否则继续判断该点处的棋子是否为己方棋子。若是则该点不可走，否则该点为象的可行走点并标记为战斗点。

public class Chess\_Xiang : BaseChess

{

public override List<Vector2> CanMovePoints(Dictionary<GameObject, Vector2> chess2Vector, Dictionary<Vector2, GameObject> vector2Chess)

{

Vector2 currentPos = chess2Vector[gameObject];

List<Vector2> canMovePoints = new List<Vector2>();

if (gameObject.tag == "Red")

{

if (currentPos.y <= 2 && currentPos.x <= 6) //45°斜向上走

{

Vector2 value = new Vector2(currentPos.x + 2, currentPos.y + 2);

Vector2 valueEye=new Vector2(currentPos.x+1, currentPos.y+1);

JudgeMovePoint(value, valueEye,canMovePoints, vector2Chess);

}

if (currentPos.y >= 2 && currentPos.x <= 6) //-45°斜向下走

{

Vector2 value = new Vector2(currentPos.x + 2, currentPos.y - 2);

Vector2 valueEye = new Vector2(currentPos.x+1,currentPos.y-1);

JudgeMovePoint(value, valueEye,canMovePoints, vector2Chess);

}

if (currentPos.y >= 2 && currentPos.x >= 2) //-135°斜向下走

{

Vector2 value = new Vector2(currentPos.x - 2, currentPos.y - 2);

Vector2 valueEye = new Vector2(currentPos.x-1,currentPos.y-1);

JudgeMovePoint(value, valueEye,canMovePoints, vector2Chess);

}

if (currentPos.y <= 2 && currentPos.x >= 2) //135°斜向上走

{

Vector2 value = new Vector2(currentPos.x - 2, currentPos.y + 2);

Vector2 valueEye = new Vector2(currentPos.x-1,currentPos.y+1);

JudgeMovePoint(value, valueEye,canMovePoints, vector2Chess);

}

}

if (gameObject.tag == "Black")

{

if (currentPos.y <= 7 && currentPos.x <= 6) //45°斜向上走

{

Vector2 value = new Vector2(currentPos.x + 2, currentPos.y + 2);

Vector2 valueEye =new Vector2(currentPos.x+1,currentPos.y+1);

JudgeMovePoint(value, valueEye,canMovePoints, vector2Chess);

}

if (currentPos.y >= 7 && currentPos.x <= 6) //-45°斜向下走

{

Vector2 value = new Vector2(currentPos.x + 2, currentPos.y - 2);

Vector2 valueEye = new Vector2(currentPos.x+1,currentPos.y-1);

JudgeMovePoint(value, valueEye,canMovePoints, vector2Chess);

}

if (currentPos.y >= 7 && currentPos.x >= 2) //-135°斜向下走

{

Vector2 value = new Vector2(currentPos.x - 2, currentPos.y - 2);

Vector2 valueEye = new Vector2(currentPos.x-1,currentPos.y-1);

JudgeMovePoint(value, valueEye,canMovePoints, vector2Chess);

}

if (currentPos.y <= 7 && currentPos.x >= 2) //135°斜向上走

{

Vector2 value = new Vector2(currentPos.x - 2, currentPos.y + 2);

Vector2 valueEye = new Vector2(currentPos.x-1,currentPos.y+1);

JudgeMovePoint(value, valueEye,canMovePoints, vector2Chess);

}

}

return canMovePoints;

}

/// 相/象专属判断是否可以走这个点

/// <param name="value"></param>

/// <param name="valueEye">象眼位置</param>

void JudgeMovePoint(Vector2 value, Vector2 valueEye, List<Vector2> canMovePoints, Dictionary<Vector2, GameObject> vector2Chess)

{

if (GameUtil.IsInChessBoard(valueEye))

{

if (vector2Chess.ContainsKey(valueEye))//若象眼位置存在棋子

return;

else

{

if (GameUtil.IsInChessBoard(value)) //若网格存在，即在棋盘内

{

if (vector2Chess.ContainsKey(value))

{

GameObject otherChess = vector2Chess[value];

if (otherChess.tag != gameObject.tag)

canMovePoints.Add(value);

}

else

canMovePoints.Add(value);

}

}

}

}

}

卒/兵：红方为“兵”，黑方为“卒”。兵（卒）只能向前走，不能后退，在未过河前，不能横走。过河以后还可左、右移动，但也只能一次一步，即使这样，兵（卒）的威力也大大增强，故有“小卒过河顶大车”之说。以红方兵为例，兵的行走点分前、左、右三个方向进行统计。首先，以向前方向为例，根据兵目前所处位置(x,y)判断兵是否已经走到敌方底线即y值是否等于9。若是则前方向没有可走点，否则继续判断y+1处是否存在棋子。若不存在则该点为兵的可行走点，否则继续判断该点处棋子是否为己方棋子。若是则该点不可走，否则该点为兵的可行走点并标记为战斗点。其次，以向左方向为例(向右同理)，根据兵目前所处位置(x,y)判断兵是否已经跨越河界即y值是否等于4。若不是则左方向没有可走点，否则继续判断x-1处是否存在棋子。若不存在则该点为兵的可行走点，否则继续判断该点处棋子是否为己方棋子。若是则该点不可走，否则该点为兵的可行走点并标记为战斗点。

public class Chess\_Zu : BaseChess

{

public override List<Vector2> CanMovePoints(Dictionary<GameObject, Vector2> chess2Vector, Dictionary<Vector2, GameObject> vector2Chess)

{

Vector2 currentPos = chess2Vector[gameObject];

List<Vector2> canMovePoints = new List<Vector2>();

//若是红方且过了河 或是 黑方且过了河，就能左右走

if ((gameObject.tag == "Red" && currentPos.y >= 5) ||

(gameObject.tag == "Black" && currentPos.y <= 4))

{

Vector2 valueLeft = new Vector2(currentPos.x - 1, currentPos.y);

JudgeMovePoint(valueLeft, canMovePoints, vector2Chess);

Vector2 valueRight = new Vector2(currentPos.x + 1, currentPos.y);

JudgeMovePoint(valueRight, canMovePoints, vector2Chess);

}

if (gameObject.tag == "Red")//若是红方，只能向上走。

{

Vector2 valueUp = new Vector2(currentPos.x, currentPos.y + 1);

JudgeMovePoint(valueUp, canMovePoints, vector2Chess);

}

if (gameObject.tag == "Black")//若是黑方，只能向下走

{

Vector2 valueDown = new Vector2(currentPos.x, currentPos.y - 1);

JudgeMovePoint(valueDown, canMovePoints, vector2Chess);

}

return canMovePoints;

}

/// 兵专属判断是否可以走这个点

void JudgeMovePoint(Vector2 value, List<Vector2> canMovePoints, Dictionary<Vector2, GameObject> vector2Chess)

{

if (GameUtil.IsInChessBoard(value)) //若网格存在，即在棋盘内

{

if (vector2Chess.ContainsKey(value))

{

GameObject otherChess = vector2Chess[value];

if (otherChess.tag != gameObject.tag)

canMovePoints.Add(value);

}

else

canMovePoints.Add(value);

}

}

}

帅/将：红方为“帅”，黑方为“将”。帅和将是棋中的首脑，是双方竭力争夺的目标。它只能在"九宫"之内活动，可上可下，可左可右，每次走动只能按竖线或横线走动一格。帅与将不能在同一直线上直接对面，否则走方判负。以红方帅为例，帅的行走点分上、下、左、右四个方向进行统计。以向左方向为例，根据帅目前所处位置(x,y)判断x-1处是否在九宫格内。若不是则该点不可走，否则继续判断该点处y方向是否有且仅有一个棋子且是敌方将。若是则该点不可走，否则继续判断该点是否存在棋子。若不存在则该点为帅的可行走点，否则继续判断该点处棋子是否为己方棋子。若是则该点不可走，否则该点为帅的可行走点并标记为战斗点。

public class Chess\_Boss : BaseChess

{

/// <summary>

/// 检测是否被将军

/// </summary>

public void DetectBeAttacked()

{

Vector2 currentPos = GameCache.Chess2Vector[gameObject];

List<Vector2> enemyPoints;

if (gameObject.tag == "Red")

enemyPoints = GameUtil.getTeamMovePoints(PoolManager.work\_List, "Black", GameCache.Chess2Vector, GameCache.Vector2Chess);

else

enemyPoints = GameUtil.getTeamMovePoints(PoolManager.work\_List, "Red", GameCache.Chess2Vector, GameCache.Vector2Chess);

foreach(Vector2 point in enemyPoints)

{ //敌军可走点有此位置点,则说明在将军

if (GameUtil.CompareVector2(currentPos, point))

{

chessSituationState = ChessSituationState.BeAttacked;

break;

}

}

}

/// 重写被点击方法，规定将帅不可以加属性

public override void ChesseClicked()

{

if (GameController.playing == Playing.BlackAdding || GameController.playing == Playing.RedAdding)

return;

base.ChesseClicked();

}

public override List<Vector2> CanMovePoints(Dictionary<GameObject, Vector2> chess2Vector, Dictionary<Vector2, GameObject> vector2Chess)

{

Vector2 currentPos = chess2Vector[gameObject];

List<Vector2> canMovePoints = new List<Vector2>();

if (gameObject.tag == "Red")

{

if (currentPos.y <= 1) //可向上走

{

Vector2 value = new Vector2(currentPos.x, currentPos.y + 1);

JudgeMovePoint(value,canMovePoints,gameObject,chess2Vector,vector2Chess);

}

if (currentPos.y >= 1) //可向下走

{

Vector2 value = new Vector2(currentPos.x, currentPos.y - 1);

JudgeMovePoint(value,canMovePoints,gameObject,chess2Vector,vector2Chess);

}

}

if (gameObject.tag == "Black")

{

if (currentPos.y <= 8) //可向上走

{

Vector2 value = new Vector2(currentPos.x, currentPos.y + 1);

JudgeMovePoint(value,canMovePoints,gameObject,chess2Vector,vector2Chess);

}

if (currentPos.y >= 8) //可向下走

{

Vector2 value = new Vector2(currentPos.x, currentPos.y - 1);

JudgeMovePoint(value,canMovePoints,gameObject,chess2Vector,vector2Chess);

}

}

if (currentPos.x <= 4) //可向右走

{

Vector2 value = new Vector2(currentPos.x + 1, currentPos.y);

JudgeMovePoint(value,canMovePoints,gameObject,chess2Vector,vector2Chess);

}

if (currentPos.x >= 4) //可向左走

{

Vector2 value = new Vector2(currentPos.x - 1, currentPos.y);

JudgeMovePoint(value,canMovePoints,gameObject,chess2Vector,vector2Chess);

}

return canMovePoints;

}

/// 帅/将专属判断是否可以走这个点

void JudgeMovePoint(Vector2 value, List<Vector2> canMovePoints, GameObject self, Dictionary<GameObject, Vector2> chess2Vector, Dictionary<Vector2, GameObject> vector2Chess)

{

GameObject enemyBoss;

if (self == createManager.GetRedBoss())

enemyBoss = createManager.GetBlackBoss();

else

enemyBoss = createManager.GetRedBoss();

if (GameUtil.IsInChessBoard(value))

{

bool existOtherChessOnSame\_X\_Axis = false;

if (value.x == chess2Vector[enemyBoss].x)

{

float enemyBoss\_Y = chess2Vector[enemyBoss].y;

if (enemyBoss == createManager.GetBlackBoss())

{

for (int i = (int)value.y + 1; i < enemyBoss\_Y; i++)

if (vector2Chess.ContainsKey(new Vector2(value.x, i)))

existOtherChessOnSame\_X\_Axis = true;

}

else if (enemyBoss == createManager.GetRedBoss())

{

for (int i = (int)value.y - 1; i > enemyBoss\_Y; i--)

if (vector2Chess.ContainsKey(new Vector2(value.x, i)))

existOtherChessOnSame\_X\_Axis = true;

}

if (existOtherChessOnSame\_X\_Axis)

{

if (vector2Chess.ContainsKey(value))

{

GameObject otherChess = vector2Chess[value];

if (otherChess.tag != gameObject.tag)

canMovePoints.Add(value);

}

else

canMovePoints.Add(value);

}

}

else

{

if (vector2Chess.ContainsKey(value))

{

GameObject otherChess = vector2Chess[value];

if (otherChess.tag != gameObject.tag)

canMovePoints.Add(value);

}

else

canMovePoints.Add(value);

}

}

}

public override void Killed(GameObject chess)

{

base.Killed(chess);

if (chess == gameObject)

{

GameController.gameStatus = GameStatus.End;

BossKilledEventWithParam(gameObject.tag);

BossKilledEvent();

}

}

/// 增加监听检测被将军事件

public override void SubscribeEvents(GameObject chess)

{

if (chess == gameObject)

{

GameController.UpdateGameDataCompleteEvent+=DetectBeAttacked;

base.SubscribeEvents(chess);

}

}

public override void CancelSubscribeEvents(GameObject chess)

{

if (chess == gameObject)

{

GameController.UpdateGameDataCompleteEvent-=DetectBeAttacked;

base.CancelSubscribeEvents(chess);

}

}

}

* 1. **战斗实现**

战斗实现需要三个步骤。首先是属性的配置，其次是属性的读取，最后是属性的计算比较。

首先，属性的配置，为了方便测试和修改数据致游戏平衡合理化，统一写在一个txt文件内，语法格式为JSon。

{

"Attributes": [

{"name": "Boss", "hp": 1, "attack": 99999, "defence": 0},

{"name": "Ju", "hp": 200, "attack": 50, "defence": 40},

{"name": "Ma", "hp": 180, "attack": 40, "defence": 25},

{"name": "Pao", "hp": 180, "attack": 45, "defence": 20},

{"name": "Shi", "hp": 100, "attack": 30, "defence": 10},

{"name": "Xiang", "hp": 100, "attack": 30, "defence": 10},

{"name": "Zu", "hp": 100, "attack": 20, "defence": 20}

]

}

接着，属性的读取，需要将txt文件里的json数据解析成AttrBox类对象。具体做法是先将资源目录里txt文件的字符串读取出来，然后通过Unity解析JSon的API反序列化存储到类或结构体当中。

public class ChessConfig

{

public static List<ChessCfg> cfg;

public static Dictionary<string, List<int>> chessDic; //名字与属性的映射

[Serializable]

public class ChessCfg //配置的各个字段

{

public string name;

public int hp;

public int attack;

public int defence;

}

[Serializable]

public class CfgClass

{

public List<ChessCfg> Attributes;

}

/// 读取Json数据

public static List<ChessCfg> ReadJson()

{

if (cfg == null)

{

TextAsset textAsset = new TextAsset();

textAsset = Resources.Load("Json/t\_ChessConfig") as TextAsset;

string txt = textAsset.text;

if (txt != string.Empty)

{

CfgClass item = JsonUtility.FromJson<CfgClass>(txt);

cfg = item.Attributes; //获取类的对象拥有的属性列表

}

}

return cfg;

}

public static Dictionary<string, List<int>> GetChessDic()

{

if (chessDic == null)

{

chessDic = new Dictionary<string, List<int>>();

cfg = ReadJson();

foreach(ChessCfg cc in cfg)

{

List<int> attrList = new List<int>();

attrList.Add(cc.hp); //attrList索引为0

attrList.Add(cc.attack); //attrList索引为1

attrList.Add(cc.defence); //attrList索引为2

chessDic.Add(cc.name, attrList);

}

}

return chessDic;

}

/// 获取某棋子初始属性值列表

public static List<int> GetAttrList(string chessName)

{

chessDic = GetChessDic();

return chessDic[chessName];

}

}

然后，前面提到的每个棋子内声明的AttrBox属性就是存储这些属性信息的容器。这样，每个棋子一开始就拥有初始属性了。

public enum AttrType

{

Hp = 0,

Attack = 1,

Defence = 2,

}

public class AttrBox : MonoBehaviour

{

private int hp;

private int attack;

private int defence;

public int Hp { get { return hp; } set { hp = value; } }

public int Attack { get { return attack; } set { attack = value; } }

public int Defence { get { return defence; } set { defence = value; } }

public int Combat

{

get

{

return GameUtil.CalCombat(Hp, Attack, Defence);

}

}

public void SetAttrList(List<int> attrList)

{

Hp = attrList[(int)AttrType.Hp];

Attack = attrList[(int)AttrType.Attack];

Defence = attrList[(int)AttrType.Defence];

}

}

public abstract class BaseChess : MonoBehaviour

{

public AttrBox attrBox;

public virtual void Awake()

{

attrBox = gameObject.GetComponent<AttrBox>();

attrBox.SetAttrList(ChessConfig.GetAttrList(chessName));

}

}

最后，战斗属性计算比较的实现算法。首先要获知战斗双方棋子身份，故要将进攻方和防守方棋子完整信息传入方法内进行处理。然后根据分析和计算最终返回出阵亡者棋子，因为返回出来的信息会被资源重用模块利用并回收，同时也会被数据层模块记录起阵亡者身份、阵亡步数、阵亡时属性等信息。

public static GameObject Battle(GameObject attacker, GameObject defender)

{

AttrBox a = GetChessAttrList(attacker);

AttrBox b = GetChessAttrList(defender);

int a\_damage=a.Attack<=b.Defence? 1 : a.Attack-b.Defence; //a对b造成的伤害

int b\_damage=b.Attack<=a.Defence? 1 : b.Attack-a.Defence;//b对a造成的伤害

int a\_times = b.Hp / a\_damage == 0 ? 1 : b.Hp / a\_damage; //a打死b所需回合数

int b\_times = a.Hp / b\_damage == 0 ? 1 : a.Hp / b\_damage; //b打死a所需回合数

if (a\_times <= b\_times)

{

a.Hp -= (b\_damage \* (a\_times - 1));

b.Hp = 0;

return defender;

}

else

{

b.Hp -= a\_damage \* b\_times;

a.Hp = 0;

return attacker;

}

}

* 1. **时间管理**

当轮到红方走棋时，红方开始计时。走完后，轮到红方加属性，红方继续计时，直到加完属性为止。黑方同理。当总局时小于规定步时时，步时等于总局时，直到总局时为0时，判为负。

public class TimeManager : MonoBehaviour

{

private static TimeManager instance = null;

public static TimeManager Instance { get { return instance; } }

private static float allTime;

private static float stepTime;

public static float r\_AllTime = 0.0f; //红局时，单位s

public static float b\_AllTime = 0.0f;

public static float r\_StepTime = 0.0f; //红步时，单位s

public static float b\_StepTime = 0.0f;

public static event GameOverEventHandlerWithParam TimeUpEventWithParam;

public static event GameOverEventHandler TimeUpEvent;

void Awake()

{

if (instance == null)

instance = this;

allTime = 60.0f \* 20.0f;

stepTime = 60.0f \* 1.5f;

r\_AllTime = allTime;

b\_AllTime = allTime;

r\_StepTime = stepTime;

b\_StepTime = stepTime;

Scene3\_UI.AddAttrCompleteEvent += ResetStepTime;

}

void Update ()

{

if (GameController.gameStatus == GameStatus.Going)

{

if (GameController.playing == Playing.OnRed || GameController.playing == Playing.RedAdding)

{

if (r\_StepTime > 0)

r\_StepTime -= Time.deltaTime;

else

{

GameController.gameStatus = GameStatus.End;

TimeUpEventWithParam("Red");

TimeUpEvent();

r\_StepTime = 0;

}

if (r\_AllTime > 0)

r\_AllTime -= Time.deltaTime;

else

{

GameController.gameStatus = GameStatus.End;

TimeUpEventWithParam("Red");

TimeUpEvent();

r\_AllTime = 0;

}

}

else if (GameController.playing == Playing.OnBlack || GameController.playing == Playing.BlackAdding)

{

if (b\_StepTime > 0)

b\_StepTime -= Time.deltaTime;

else

{

GameController.gameStatus = GameStatus.End;

TimeUpEventWithParam("Black");

TimeUpEvent();

b\_StepTime = 0;

}

if (b\_AllTime > 0)

b\_AllTime -= Time.deltaTime;

else

{

GameController.gameStatus = GameStatus.End;

TimeUpEventWithParam("Black");

TimeUpEvent();

b\_AllTime = 0;

}

}

}

}

/// 重置局时与步时

public static void ResetAllTime()

{

r\_AllTime = allTime;

b\_AllTime = allTime;

ResetStepTime();

}

/// 重置步时

public static void ResetStepTime()

{//当局时小于规定步时，那么步时就等于局时

r\_StepTime = r\_AllTime < stepTime ? r\_AllTime : stepTime;

b\_StepTime = b\_AllTime < stepTime ? b\_AllTime : stepTime;

}

}

* 1. **悔棋和复盘**

悔棋和复盘都需要实现记录棋谱和根据棋谱复原棋盘的功能，两者的区别只是复原到的步数不同而已。

首先是棋谱的记录，每一步都需要记录各棋子的位置数据和属性数据。在GameCache.cs脚本内声明了这两个存储变量，并提供了相应的存取方法。

public class GameCache

{

/// 记录每回合的所有棋局图谱信息

public static List<Dictionary<GameObject, Vector2>> maps;

/// 记录每回合的所有棋子对应的属性信息

public static List<Dictionary<GameObject, string>> attrMaps;

/// 添加棋谱

public static void SetMaps()

{

if (maps == null)

maps = new List<Dictionary<GameObject, Vector2>>();

Dictionary<GameObject, Vector2> temp = new Dictionary<GameObject, Vector2>();

foreach (KeyValuePair<GameObject, Vector2> kvp in Chess2Vector)

{

//一定要这样遍历赋值的，否则如果直接temp=CalculateUtil.chess2Vector的话

//就相当于引用了这个静态字典，每次添加到棋谱里就是同一个temp数据

temp.Add(kvp.Key, kvp.Value);

}

maps.Add(temp); //添加棋谱

}

/// 添加每回合全部棋子对应属性信息图谱

public static void SetAttrMaps()

{

if (attrMaps == null)

attrMaps = new List<Dictionary<GameObject, string>>();

Dictionary<GameObject, string> dic=new Dictionary<GameObject,string>();

foreach (GameObject chess in PoolManager.work\_List)

{

AttrBox chessAttrs = GameUtil.GetChessAttrList(chess);

string attrStr=chessAttrs.Hp+"\_"+chessAttrs.Attack+"\_"+chessAttrs.Defence;

dic.Add(chess, attrStr);

}

attrMaps.Add(dic);

}

}

然后，根据记录的棋谱复原相应的棋子位置。

public class GameUtil

{

/// 还原某个棋子在某个棋局的位置

public static void ResetChessByMaps(GameObject chess, Vector2 point)

{

if (chess.transform.parent != PoolManager.workChesses)

PoolManager.Take(chess); //被吃的棋子从回收区提取回来

int x = (int)point.x;

int y = (int)point.y;

chess.transform.position = Scene3\_UI.cells[x, y].transform.position;

}

/// 根据步数设置棋盘图谱

public static void SetChessBoardByMaps(int step)

{

Dictionary<GameObject, Vector2> targetMap = GameCache.maps[step];

foreach(KeyValuePair<GameObject,Vector2> kvp in targetMap)

ResetChessByMaps(kvp.Key, kvp.Value); //还原所有棋子的位置

foreach(KeyValuePair<GameObject,int> kvp in GameCache.LoserStepDic)

if (kvp.Value <= step) //在此步前阵亡的棋子都回收起来

PoolManager.Restore(kvp.Key);

Dictionary<GameObject,string> targetAttrMap=GameCache.attrMaps[step];

foreach (KeyValuePair<GameObject, string> kvp in targetAttrMap)

{

int[] arr = StrAttr2IntArr(kvp.Value);

GetChessAttrList(kvp.Key).Hp = arr[0]; //还原所有棋子的属性

GetChessAttrList(kvp.Key).Attack = arr[1];

GetChessAttrList(kvp.Key).Defence = arr[2];

}

}

}

另外，复盘还需要显示每一步有属性变化的棋子情况，这就还需要记录战斗阵亡者与对应步数的映射变量和比较相邻两步属性差异的方法。

public class GameCache

{

/// 记录阵亡者与对应步数的映射

private static Dictionary<GameObject, int> loserStepDic;

public static void SetLoserStepDic(GameObject loser, int step)

{

if (loserStepDic == null)

loserStepDic = new Dictionary<GameObject, int>();

loserStepDic.Add(loser, step);

}

public static Dictionary<GameObject, int> LoserStepDic

{

get { return loserStepDic; }

}

}

public class GameUtil

{

/// <summary>

/// 比较此步与前一步所有棋子的属性差异

/// </summary>

/// <param name="step"></param>

/// <returns>最多返回两句描述，因为两步间最多存在两棋子属性有差异</returns>

public static string[] CompareStepAttr(int step)

{

if (step == 0)

return new string[] { "", "" };

string[] desc = new string[2];

desc[0] = "";

desc[1] = "";

int index = 0;

Dictionary<GameObject, string> curAttrMap = GameCache.attrMaps[step];

Dictionary<GameObject,string> preAttrMap=GameCache.attrMaps[step-1];

foreach (KeyValuePair<GameObject, string> kvp in curAttrMap)

{

string preAttrStr = preAttrMap[kvp.Key];

if (kvp.Value != preAttrStr)//属性有差异且此步前不在阵亡者列表中

{

if (GameCache.LoserStepDic.ContainsKey(kvp.Key)

&& GameCache.LoserStepDic[kvp.Key] <= step)

continue;

desc[index] = GetChineseChessName(kvp.Key);

int[] curArr = StrAttr2IntArr(kvp.Value);

int[] preArr = StrAttr2IntArr(preAttrStr);

int deltaHp = curArr[0] - preArr[0];

int deltaAttack = curArr[1] - preArr[1];

int deltaDefence = curArr[2] - preArr[2];

if (deltaHp > 0)

desc[index] = desc[index] + " " + "生命提升" + deltaHp;

else if (deltaHp<0)

desc[index]=desc[index]+" "+"生命损失"+Mathf.Abs(deltaHp);

if (deltaAttack > 0)

desc[index] = desc[index] + " " + "攻击提升" + deltaAttack;

if (deltaDefence > 0)

desc[index] = desc[index] +" " +"防御提升" + deltaDefence;

index++;

}

}

return desc;

}

}

# 总结

* 1. **总结**

本系统以开发一款完整的、可供用户体验的、带有创新性的游戏软件为目标进行缜密性开发。目前，决战象棋游戏的研发还处于第一版，目前可以完整的体验到游戏的各个功能。供玩家操作的核心功能有：走棋、加属性、再来一局、悔棋和复盘。这些核心功能的实现得益于数据层框架的完整搭建，有了存储数据的载体和提供修改数据的方法，这些功能才能一一实现。实现逻辑较简单的可供玩家操作的其他功能还有：返回、求和、认输。

这款游戏最主要的创新在于加入属性战斗模块，双方棋子通过战斗生存来简化棋局。其中，属性的配置数据以Json格式存储为txt文件，方便日后对数据合理化调整，对游戏的可维护性提供了一种可能。

然后通过项目我还得到了一些关于代码的经验，就是正常的软件开发应该是懒惰式开发，也称为忍耐式开发，这种开发的表现就是在写代码之前，代码员需要花费大量的时间去考虑所有可能的解决方案和途径。先把问题理解清楚，把解决的问题解决了，再把架构梳理好，以这样的方式坚持下去，程序员会得到非常大的提高。每个程序员都会有自己的书写风格和思维逻辑，项目工程的进行也不一样，但是没有规矩不成方圆，继续学习并使用软件工程和一些编程的思维，使项目的进度和代码的风格都有所提高和改进。

* 1. **技术难点与解决**

**难点一：**棋子在移动时保证相关数据的映射关系精确无误。棋子以可走点位置为目标在移动完后，需要知道它所在的二维坐标。因为有存储固定的三维坐标和二维坐标的映射Dictionary<Vector3, Vector2> coords，所以按正常来说以棋子的三维坐标为key是可以访问对应的二维坐标的。但是，访问失败了，抛出的错误为没有对应的key值。

**问题所在：**后来发现，是Unity的Itween运动插件导致的。比如可走点三维坐标为(100,50,0)，对应二维坐标为(2,3)，当棋子以此可走点的三维坐标为目标移动完后，棋子的三维坐标有可能是(100,50.0001,0)。导致以棋子三维坐标为key访问对应的二维坐标失败了，因为确实不存在三维坐标为(100,50.0001,0)的key。

**解决思路：**将相关数据强制转换为整形，消除微弱差别的影响。

**解决过程：**每次棋子移动完后，在更新棋局映射时，分别对棋子x,y,z三个坐标变量强制为整形，再添加入映射。

for (int i = 0; i < chessList.Count; i++)

{

Vector3 pos = chessList[i].transform.position;

Vector3 v3 = new Vector3((int)pos.x, (int)pos.y, (int)pos.z);//强制为整形

chess2Vector.Add(chessList[i], coords[v3]);

vector2Chess.Add(coords[v3], chessList[i]);

}

**难点二：**切换场景时保证相关数据完全清空。在切换场景时，经常抛出各种空引用异常和一些数据仍有保留的问题。

**问题所在：**后来发现，场景切换时销毁了所有物体。但在主场景中每个地方添加订阅的事件引用仍然存在，或者对棋子对象操作的引用也依然存在，一些游戏状态的变量也依旧保留着。

**解决思路：**在切换场景或销毁物体时调用的方法内清空数据。

**解决过程：**在各个脚本内添加OnDestory函数，当物体销毁时，Unity会自动调用这个函数，因此在此函数内可以取消订阅相关事件。然后在切换场景时，在OnSceneLoaded函数内重置相关游戏状态数据。

**难点三：**复盘时，需要获取当前步与前一步所有棋子的属性差异并显示出来。需要过滤掉阵亡棋子的信息。

**解决思路：**存储阵亡者与对应步数的信息，再遍历比较相邻两步棋子属性差异。

**解决过程：**在添加棋谱时，顺便检测当前是否发生过战斗，若发生了就添加阵亡者和对应步数的映射关系。有了这个关系后，在复盘时，每次遍历相邻两步间的所有棋子差异前，首先判断棋子是否存在于这个映射关系中，若不存在说明此棋子在当前步还未阵亡，可以往下比较属性。若存在说明棋子是阵亡的，要再继续判断棋子阵亡的步数是否小于等于当前步，若是则过无需比较属性，过滤掉此棋子，否则就是在未来才阵亡的，要继续比较属性。

* 1. **展望**

目前这一版的决战象棋游戏是可以完整体验的一个版本，但我希望不仅于此，我还打算在未来拓展更多的新功能。在未来，希望可以实现决战象棋的AI功能，这不仅需要借鉴博弈论和普通象棋AI算法，还要结合考虑战斗属性比较的因素，改进甚至重组出属于决战象棋的AI功能。希望还可以实现棋盘可走点衰减功能，即缩小安全区的功能，这需要考虑棋盘可走点映射的相关移除带来的一系列影响。总之，我希望我的想法在未来可以一一实现，打造出全新的游戏，给用户带来不一样的游戏体验。

# 参考文献

[1] 简书. unity3d配置Android环境，打包发布Apk流程详解[OL]. https://www.jianshu.com/p/3c67fbfbb67c,2016.4.2

[2] CN博客. [整理]unity3d优化总结篇[OL]. http://www.cnblogs.com/harlan1009/p/3970584.html,2014.9.14

[3] 简书. unity5.3新API使用进度条,异步加载场景的代码[OL]. https://www.jianshu.com/p/bf4ec64cb454,2014.9.14

[4] CSDN. C#中委托、事件和回调函数的理解[OL]. https://blog.csdn.net/hit\_why/article/details/54604908,2017.1.18

[5] 百度文库. 计算机专业毕业论文(象棋游戏)[DB]. https://wenku.baidu.com/view/782c1d1910a6f524ccbf85c2.html,2010.12.29

[6] 百度文库. 中国象棋游戏设计与实现毕业设计毕业论文[DB]. https://wenku.baidu.com/view/037039115b8102d276a20029bd64783e09127d69.html?rec\_flag=default&mark\_pay\_doc=1&mark\_rec\_page=1&mark\_rec\_position=5&mark\_rec=view\_r\_1&clear\_uda\_param=1.2017.10.9

[7] Unity. Unity - Scripting API[OL]. https://docs.unity3d.com/560/Documentation/ScriptReference/index.html,2017.7.12

[8] 简书. Unity基础 -- Git版本的源代码管理工具使用[OL]. https://www.jianshu.com/p/fa06a5004d06,2016.12.15

[9] iTween. iTween for Unity by Bob Berkebile[OL]. http://www.pixelplacement.com/itween/documentation.php,2017.1.18

[10] 百度百科. 中国象棋百度介绍[OL].

https://baike.baidu.com/item/%E4%B8%AD%E5%9B%BD%E8%B1%A1%E6%A3%8B/278314?fr=aladdin,2017.1.18

# 致谢

时光穿梭，大学生活即将结束，四年校园生活，学校及学院的老师一直都在悉心教导、培养我的专业技能、专业知识，使自己在专业知识、能力上打下了良好的基础，在此向辛勤付出的老师们表示感谢。同时，需要感谢刘志煌老师对本次毕业设计的悉心指导和帮助。最后，最重要的是感谢父母的养育之恩，尤其是在大学四年间对我的每一个重大决定和行动的支持与帮助。

即将毕业，我走向游戏行业，在系统的开发和毕业论文的写作过程中，我对游戏开发的流程更加了解。我也非常有信心能在游戏行业内干出一番业绩，成就更美好的人生。