国际象棋软件开发文档

制作小组成员：张朕银、陈煜昊、易晓玲、王依婷、王贵成

软件名称：国际象棋

所属类别：棋类游戏

编写日期：2019.6

目录

一.设计目标

1.1 功能概述

国际象棋作为国际通行棋种，也是一项智力竞技运动，之前只有二人对弈形式，如今借助计算机强大的计算能力，可以开发人机对战的国际象棋程序。另外，通过国际象棋游戏程序这一形式，也大大便利了人们把国际象棋作为业余时间的有效娱乐方式，因而大大促进了国际象棋的普及。从这几点上来看，国际象棋程序的开发与算法的研究具有非常重大的现实意义。

在本软件的开发过程中，软件的功能需要满足以下几点要求：

（1）在windows平台下完成软件开发，其中图形库需采用cocos2d，软件开发的编程语言为C++。

（2）开发工具要求：cocos2d-x-3.17.1及以上版本， Visual Stdio 2019及以上版本。

（3）具有双人对战，人机对战，双机对战三种模式，其中人机对战和双机对战需对电脑AI进行设计。

（4）构建合适的图形界面，使得程序对用户具有较好的友好性。

（5）程序必须可以自动检查走子有效性，自动判断胜负。

1.2 产品概述

此软件作为复旦大学微电子学院微电子科学与工程专业选修课程《软件设计与开发》的课程设计，围绕 C++语言，运用 C++相关知识，设计并实现一个具有图形界面的小游戏。以期能够综合利用课程所学内容，熟练并掌握所学知识，体验软件开发流程，增强 C++语言的编写能力。

本国际象棋游戏支持用户在计算机本地进行国际象棋游戏，该游戏支持单人，双人两种模式。单人满足人机对战，而双人满足双人对战。双机对战则无需人参与，电脑自动进行双机交替下棋。其中，对计算机行棋设定了难度分级，从而能满足不同等级选手练习需要。

1.3 功能列表与解说

在本软件中，用户可以通过使用鼠标对图形界面进行操作，进行国际象棋的游戏。

游戏进行时的界面如下图：

游戏中应实现下列功能：

（1）游戏设置：

本国际象棋游戏采用多个场景设置。

1）游戏一开始的场景：在界面正中间有三个按钮：New Game、Settings 和 Exit。点击 New Game 进入新游戏，点击 Settings 进入设置环节（仍在开发中），点击 Exit 退出游戏。

2）在 1 开始界面点击 New Game 进入维度选择界面，点击 2D 进入二维模式，点击 3D 进入三维模式，点击 Back 回退到上一界面。（其中 3D 模式仍在开发中）

3）在 2 维度选择界面中点击 2D 或 3D 后进入难度选择界面，共有 5 种模式可供选择，人机对战模式中有三档难度，分别为 Easy，Medium，Hard；还有双人对战模式和双机对战模式；点击 Back 回退到上一界面。

实现思路：使用鼠标点击在多个场景之间切换，并根据返回值决定游戏的维数和难度并调用算法部分的初始化函数，访问之前的值并传给后台进行游戏设定，从后台得到初始棋子数组，根据该数组将棋子画在棋盘上。

（2）行棋：

当前本方行棋时，可以用鼠标左键点击选定一个本方棋子，此时该棋子稍稍抬起，然后左键点击目标位置，系统会判断这一步是否合法。若检测为合法一步，则当前棋子移至鼠标点击的位置；若为非法一步（鼠标点击不在棋盘上、鼠标点击处有本方棋子、鼠标点击处不是该棋子的可走路线），则当前棋子落下，此时可以继续选择其它棋子。

当选择的是双人对战模式，则通过鼠标点击，使白方、黑方依次下棋。

当选择的是双机对战模式，则通过点击pause按钮开始观看，双机对战采取自动对下模式，两机均采取hard难度模式行棋；若想暂停观看，则可以再点击一次pause按钮；点击restart按钮双机对战将重新开始；点击regret按钮可以悔一步棋（当未看清双机对战时的双方的走棋时，可以按此按钮回退），可多次点击悔棋按钮进行多步回退。

实现思路：

在鼠标左键点击某棋子时，判断该棋子是否为当前行棋方的棋子，是的话就选中该棋子。此时左键点击鼠标就会调用后台的走棋函数，走法不合法等同于取消选中棋子，否则就完成一步走棋，即将棋子移至新位置。如果有吃子，将被分别置于棋盘左右两方，按照被吃的顺序依次排布。如果是人机对战，则还要调用后台相关函数计算当前所有合法走法，再在其中选择一步作为当前 AI 的走棋。

（3） AI 走棋：

支持人机对战，轮到电脑方时能够自动产生一步走棋，当己方走完一步之后，机器将走出自己的一步。

实现思路：

调用后台的AI走棋函数，先找出当前所有合法走法，不同难度模式下AI选择的下一步走法有所区别，选出下一步走法后，电脑将自动走出一步。如果有吃子，该子将被分别置于棋盘左右两方，按照被吃的顺序依次排布。

（4）游戏结束：

游戏结束画面：游戏结束时，会弹出game over窗口，显示出哪一方游戏胜利，在窗口中可点击按钮重新开始游戏。

实现思路：每走一步调用后台函数判断游戏是否结束。若某方有王被吃，或者某方被将且无路可走，则另一方胜利，则跳出游戏结束窗口，并显示白方或者黑方胜利；若某方当前无路可走但未被将，则算和棋，此时也会跳出游戏结束窗口。在游戏结束窗口处可点击按钮重新开始游戏。

1.4 注意事项

上述功能有以下注意事项：

1、在游戏一开始界面中的setting按钮仍在开发中，此时点击setting无响应。

2、 3D模式尚在开发中，2D模式已基本测试完成，可以先选择2D模式。

3、 2-players模式下，黑白双方轮流行棋，且为保证公平性，游戏规定一定是一方下过之后才会让另一方下。

1.5 系统环境

支持平台：win10/Android/ios

二. 设计思路

2.1 实现场景转换

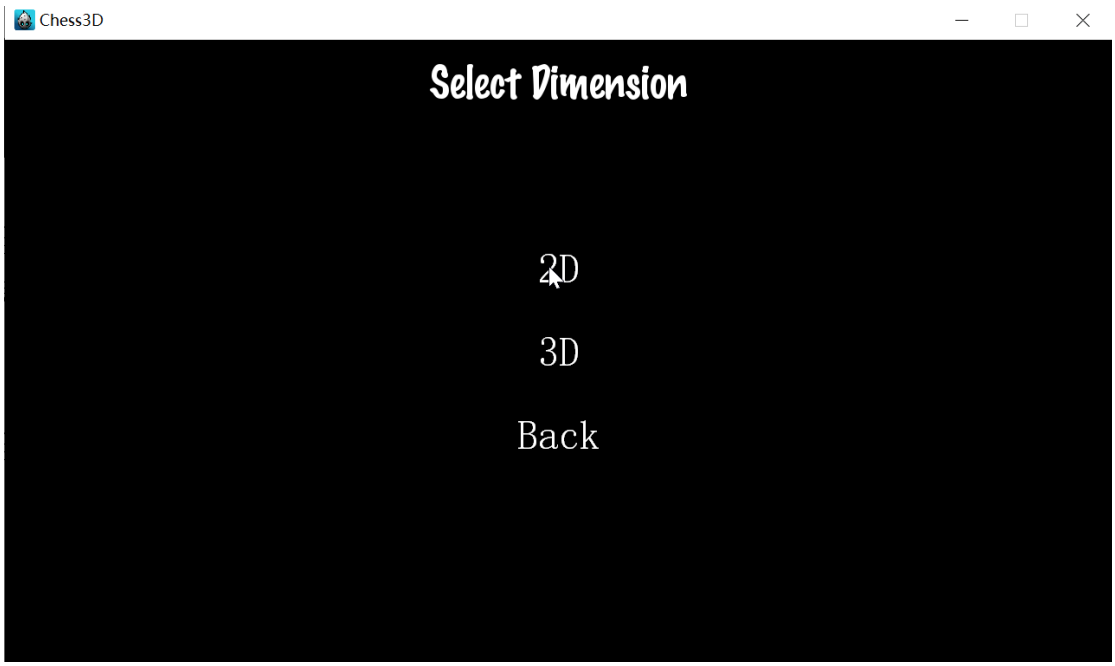
原本我们希望能够做出一款可以游玩 2D 国际象棋和 3D 国际象棋（《生活大爆炸》中的谢尔顿曾经玩过的一款棋盘与传统国际象棋有很大差异的游戏），因此需要有适当的用户引导，让用户能够一目了然地选择他想要游玩的棋种和难度。

我们小组比较喜欢分页式的用户引导，即在当前页面做出选择后就跳转到其它页面，因此为玩家主要设计了四种页面：开始界面，维度选择界面，难度选择界面，游戏界面。

在开始界面（图一），玩家可以点击 New Game 开始游戏，可以点击 Exit 离开游戏。设计中 Settings 按钮点击后会进入设置界面，可以进行音量大小设置、语言设置等，但由于只是锦上添花的内容，不影响游玩的核心功能，排在了日程的后面。后期时间紧迫便没有来得及实装。

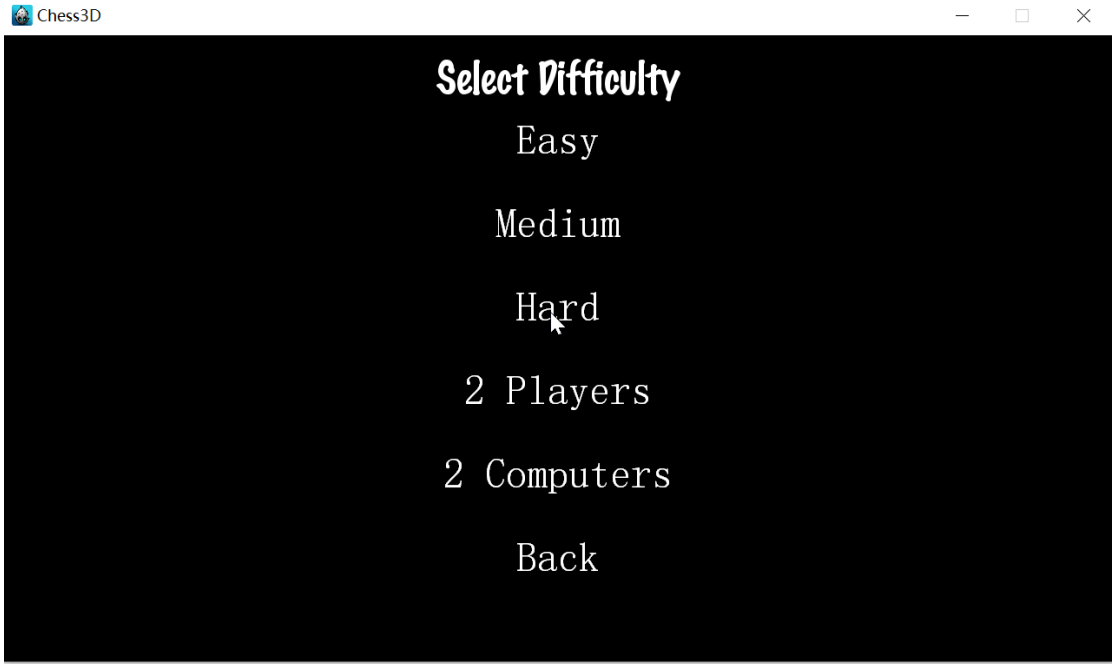
图一、开始界面

当玩家点击 New Game 时，跳转到维度选择界面（图二），可以选择 2D 国际象棋和 3D 国际象棋（待开发），也可以选择回退到开始界面。



图二、维度选择界面

当玩家选择了 2D 国际象棋或者 3D 国际象棋后，跳转到难度选择界面（图三）。难度选择界面中可以选择简单难度（电脑）、中等难度（电脑）、困难难度（电脑）、双人模式、看两台电脑对战五种模式，也可以选择回退到维度选择界面。其中看两台电脑对战的模式是之后实现的过程中新增的，一开始的设计方案中只有前四种。



图三、难度选择界面

当玩家选择了任意难度后，跳转到正式游玩界面（图四），可以根据规则进行游玩。如果玩家想要重新回到开始界面，可以点击屏幕右上角固有的菜单按钮，打开菜单界面（图五），

选择是继续游戏还是回到开始界面。

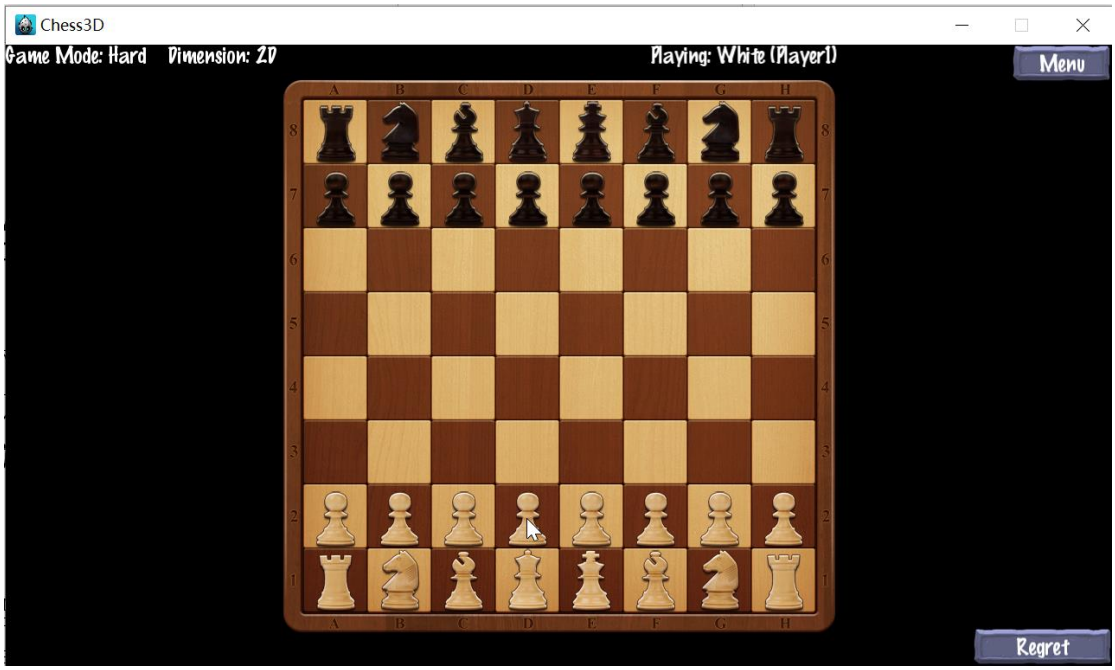
图四、游戏界面

图五、菜单界面

2.2 实现棋盘可视化

对于下棋程序，棋盘的可视化就像 HelloWorld 程序一样基础而重要，因此我们决定在在实现场景转换之后实现棋盘的可视化（图六），尽管此时棋盘可能还不具有用户行为响应功能。

实现棋盘可视化的目标为，棋盘能够正确显示，棋子能够逐一正确摆放在棋盘上。

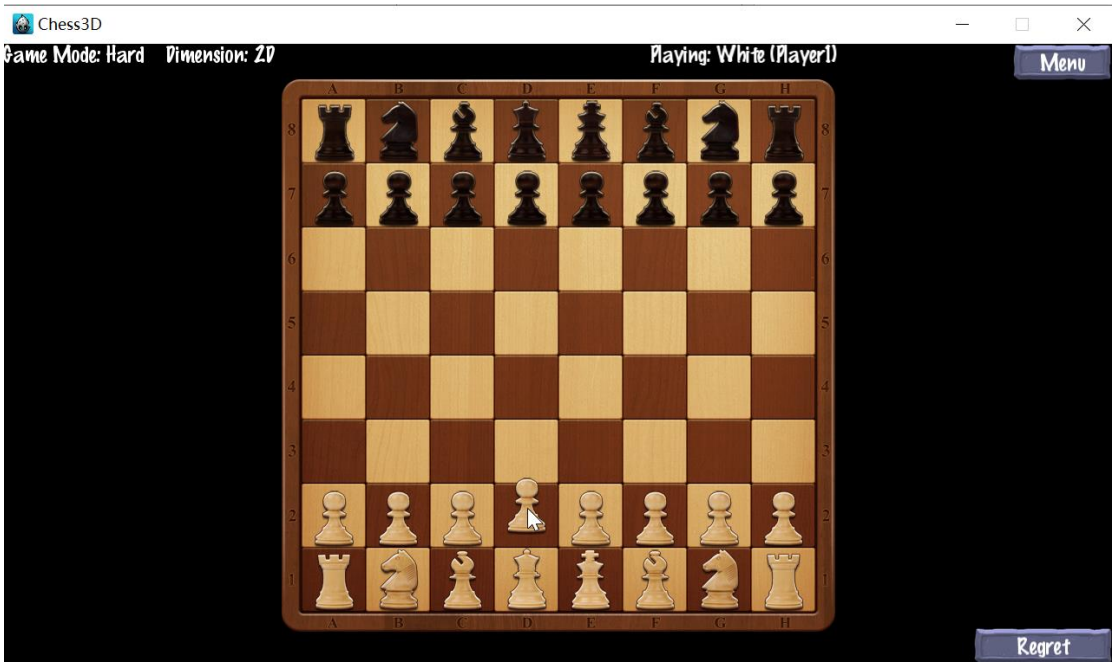


图六、棋盘的可视化

当然，简单地摆放一些图片是不够的。在实现棋盘的可视化时，应该顺便考虑棋盘相应的数据结构，以便后面的功能开发。

2.3 实现用户行为响应

当实现了棋盘的摆放后，我们小组决定应首先实现双人对战，也就是实现用户行为的响应。对于正在走的一方，他不可以移动对方的棋子；当他单击自己的棋子时，棋子会轻轻拿起（图七）；当他已经拿起自己的一个棋子时，再次单击另一区域时，如果这两次单击所代表的行为是合乎规则的，那么这一步将被走出，如果这两次单击所代表的行为是不合乎规则的，那么棋子将被放下，玩家需要尝试做出一次合乎规则的着法，着棋权才会交给对手。



图七、用户行为响应

这里会遇到一个问题：如果当前棋局下，某一方没有任何可行的走法，是否应该判负或者和棋？这个问题将在“终盘判定”中得到解决。

2.4 实现电脑自动随机走棋（Easy）

在实现用户行为响应后，我们小组认为应当实现电脑的自动走棋。为了渐进式设计，我们认为先让电脑随机走棋是较为合理的。当然，程序不会让电脑方走出不合乎规则的棋来。

这种走法将会被用于 Easy 难度。

2.5 实现应将和终局判定

在初步实现电脑自动走棋的功能后，我们小组认为应该解决一下应将和终局判定的问题。

根据国际象棋的规则，当某一方正在被将军（Check）时，必须着一子以化解这个危机。如果对于这个危机没有解法，那么这一方就算做被将死（Checkmate）。同时，当某一方没有被将军时，他不可以出一步棋，如果这步棋走过之后他就会被将军。

也就是说，当某一方行棋时，不可以走出“决定权交给对手时王会被吃掉”的棋。

也就是说，无论是玩家还是电脑，都要遵守一个比之前更加拘谨的规则。

另外，根据这个规则，当某一方行棋的时候，他可能无子可落。如果他此时正被将军，那么场面就是将死（Checkmate）。如果他此时没有被将军，但仍然无子可落，那么场面就是逼和（Stalemate）。这两种情况都会触发“游戏结束”的窗口（图八图九）。

图八、Checkmate 触发游戏结束

图九、Stalemate 触发游戏结束

2.6 实现电脑贪吃走法（Medium）

在之前实现了电脑的随机走法之后，我们发现与电脑对战时电脑非常傻。虽然他懂得规则，但经常出现送子的情况，即走了一步棋之后将子送给对方吃掉的走法。当然，这些原因都是因为它的走法是随机的。

而相应的，我们决定为电脑设计一个能够主动吃子的走法，成为贪吃走法。这种走法至少让电脑学会了吃子，虽然它可能仍然看不出一些陷阱。

这种走法将会被用在 Medium 难度。

2.7 实现悔棋按钮

在实现了上述功能后，玩家就会开始频繁遇到想要悔棋的情况了，因为 Medium 难度的电脑已经开始令人难缠，玩家在不熟悉规则的情况下可能会需要悔棋。

为了实现悔棋按钮，一个用于存放走法的历史记录是必要的。并且，在菜单打开的情况下，悔棋按钮不可以生效。

并且，悔棋按钮的内部逻辑对于之后的深度搜索也是很有必要的，因为深度搜索需要电脑自行模拟对方着某步后棋局的状况，并且还要能够将场景完全还原。

2.8 实现电脑自行对战

在准备实现电脑深度搜索算法之前，我们小组想到可以实现双机对战，一来电脑的反应

速度比人的快，二来在测试电脑的功能与性能时让人手动对弈也十分麻烦，三来也能够多一些调试的乐趣。

2.9 实现电脑深度搜索走法（Hard）

了解到棋类游戏一般都会使用到alpha-beta剪枝，我们决定应用ab剪枝作为我们的Hard难度AI算法。

2.10*前后端分离的代码重构

之所以加星号是因为这属于突发事件，并不在一开始的考虑中。

事件发生的位置大约在2.6~2.7之间，主要原因是2.7的悔棋按钮让人联想到2.9电脑深度搜索走法需要不移动前端而只在后端进行运算，我们认为前后端分离的重构过程最好早些完成。

三. 设计实现

3.1 开发库

项目的开始，我们组找到了两个较为合适的库，一个是Qt环境自带的库，另一个是Cocos2d-x（以下简称Cocos库，因为本文档中几乎不涉及cocos-creator的内容所以不至于混淆）。其中Qt编译环境需要较大的空间占用，而Cocos库则只需要根据最初生成的VS项目文件，在Visual Studio中进行编辑即可。两者都是C++环境的。另外，Qt的界面让人不是很喜欢，因此最终我们选定的开发库为Cocos库。

在此简单介绍一下Cocos库的基本类型和基本功能。详情请参见网址https://docs.cocos.com/cocos2d-x/manual/zh/ 提供的用户手册。

3.1.1 入口

这个概念只是为了迎合平常使用C++的习惯。平常使用C++的时候往往会去自己新建main()函数，或者找到已有头文件中的main()函数，并且在其中加入自己的代码。实际上这个“入口”并不需要是main()，尤其是使用cocos2d-x这样大型的库的时候。

观察已经生成的cocos Helloworld项目，可以找到名为AppDelegate的cpp文件与头文件，它们可以算是“入口”，使用cocos库的程序员可以根据C++语法决定如何在入口内配置自己的代码。

3.1.2 自定义场景 Scene

在AppDelegate.cpp的函数bool AppDelegate::applicationDidFinishLaunching()的末尾（大约行标120之后），可以找到如下代码：

```
// create a scene. it's an autorelease object
    auto scene = HelloWorld::createScene();
    // run
    director->runWithScene(scene);
```

其意义是，新建一个在别处定义的场景类的实体，并以此开始运行。

AppDelegate.cpp 的一开始已经包含了 HelloWorld 场景类的头文件。

```
#include "HelloWorldScene.h"
```

如果要自定义开始场景，就需要对高亮的部分进行修改。当然，头文件以及源文件的编写另谈。

相应的场景类型的头文件 HelloWorld.h 内容应当如下（仍以 HelloWorld 为例）：

```
#ifndef __HELLOWORLD_SCENE_H__
#define __HELLOWORLD_SCENE_H__
#include "cocos2d.h"
class HelloWorld : public cocos2d::Scene
{
public:
    static cocos2d::Scene* createScene();
    virtual bool init();

    // a selector callback
    void menuCloseCallback(cocos2d::Ref* pSender);

    // implement the "static create()" method manually
    CREATE_FUNC(HelloWorld);
};
#endif // __HELLOWORLD_SCENE_H__
```

高亮部分需要自定义修改并保持一致。

但仔细想想就知道这只是一般情况下自定义场景的方法，事实上如果需要定义数个相互之间存在继承关系的场景类型，就需要根据 C++知识储备进行更大幅度的修改，无需直接遵守这些规则。这些规则只是起到一个基本的引导作用。

相应的场景类型的源文件 HelloWorld.cpp 内容应当如下：

```
#include "HelloWorldScene.h"
#include "SimpleAudioEngine.h"
USING_NS_CC;
Scene* HelloWorld::createScene()
{
    return HelloWorld::create();
```

```
    }
    // Print useful error message instead of segfaulting when files are not there.
    static void problemLoading(const char* filename)
    {
        printf("Error while loading: %s\n", filename);
        printf("Depending on how you compiled you might have to add 'Resources/' in front of filenames in HelloWorldScene.cpp\n");
    }
    //on "init" you need to initialize your instance
    bool HelloWorld::init()
    {
        //////////////////////////////
        // 1. super init first
        if ( !Scene::init() )
        {
            return false;
        }
        auto visibleSize = Director::getInstance()->getVisibleSize();
        Vec2 origin = Director::getInstance()->getVisibleOrigin();

        return true;
    }
```

其中 visibleSize 和 origin 这两个变量是为了之后添加其他组件时更方便，相当于常量。

以上只是必要部分，如果真的只保留这些必要部分，那么显示出来的将会是黑屏。

3.1.3 添加纯图片（用精灵实现）

```
auto sprite = Sprite::create("HelloWorld.png");
  if (sprite == nullptr)
  {
      problemLoading("'HelloWorld.png'");
  }
  else
  {
      // position the sprite on the center of the screen
      sprite->setPosition(Vec2(visibleSize.width/2 + origin.x,  visibleSize.height/2 + origin.y));
      // add the sprite as a child to this layer
      this->addChild(sprite, 0);
  }
```

3.1.4 添加纯文字

```
auto label = Label::createWithTTF("Hello World", "fonts/Marker Felt.ttf", 24);
  if (label == nullptr)
  {
```

```
    problemLoading("'fonts/Marker Felt.ttf'");
  }
  else
  {
    // position the label on the center of the screen
    label->setPosition(Vec2(origin.x + visibleSize.width/2,
                            origin.y + visibleSize.height -  label->getContentSize().height));
    // add the label as a child to this layer
    this->addChild(label, 1);
  }
```

以上展示的组件都是没有鼠标事件响应功能的，之后会展示几种有鼠标事件响应功能的组件以及创建方法。

目前发现最推荐的方式是添加无鼠标事件响应的挂件，然后添加监视器以响应鼠标事件。

3.1.5 添加单个图片按钮（menu）

（以 cocos 自带的右下角的关机按钮为例）：

```
auto closeItem = MenuItemImage::create(
                                "CloseNormal.png",
                                "CloseSelected.png",
                                CC_CALLBACK_1(HelloWorld::menuCloseCallback,  this));
  if (closeItem == nullptr ||
    closeItem->getContentSize().width <= 0 ||
    closeItem->getContentSize().height <= 0)
  {
    problemLoading("'CloseNormal.png' and 'CloseSelected.png'");
  }
  else
  {
    float x = origin.x + visibleSize.width -  closeItem->getContentSize().width/2;
    float y = origin.y + closeItem->getContentSize().height/2;
    closeItem->setPosition(Vec2(x,y));
  }
  // create menu, it's an autorelease object
  auto menu = Menu::create(closeItem, NULL);
  menu->setPosition(Vec2::ZERO);
  this->addChild(menu, 1);
```

其中 menuCloseCallback 是回调函数，需要在别处定义。此处也可以用 lambda 函数代替，在后文“关于 CC_CALLBACK_1 函数”中有更多信息。

closeItem 是具有图像的、可显示的、具有具体位置信息的实体，而 menu 似乎只是和 closeItem 相连接的一个无形象实体。

3.1.6 添加单个文字按钮（menu）

```
auto item = MenuItemFont::create("Test Scene", [&](Ref * sender) {
        Director::getInstance()->pushScene(TestScene::create());
        });
    item->setFontSizeObj(24);
    item->setFontNameObj("fonts/Marker Felt.ttf");
    //item->setFontName("fonts/Marker Felt.ttf");
    auto test_menu = Menu::create(item, NULL);
    test_menu->setPosition(Vec2(visibleSize.width / 2 + origin.x, visibleSize.height / 2 + origin.y));
    this->addChild(test_menu, 1);
```

3.1.7 批量添加文字按钮（menu）

```
Vector<MenuItem*> MenuItems;
auto closeItem = MenuItemImage::create("CloseNormal.png", "CloseSelected.png",
                                       CC_CALLBACK_1(HelloWorld::menuCloseCallback, this));
MenuItems.pushBack(closeItem);
/* repeat for as many menu items as needed */
auto menu = Menu::createWithArray(MenuItems);
this->addChild(menu, 1);
```

一个更为省事的做法是用 lambda 函数代替此处的 CC_CALLBACK_1 回调函数。

注意：Menu 的默认位置是屏幕右上角，而 MenuItem 的默认位置是 Menu 的中央。

3.1.8 添加鼠标响应监听器

这是非常重要的一个功能。

```
auto movable = Sprite::create("movable.png");
    movable->setPosition(Vec2(visibleSize.width / 3 + origin.x,  visibleSize.height / 3 + origin.y));
    this->addChild(movable);
    auto listener = EventListenerTouchOneByOne::create();
    listener->onTouchBegan = [](Touch * t, Event * e) {
        return false;
    };
    listener->onTouchMoved = [](Touch * t, Event * e) {

    };
    listener->onTouchEnded = [](Touch * t, Event * e) {

    };
    Director::getInstance()->getEventDispatcher()->addEventListenerWithSceneGraphPriority(listener, movable);
```

看上去 EventListenerTouchOneByOne::create()所构造的监听器是针对触摸事件的（即主要应用在手机端或平板电脑等具有触摸功能的设备），但实际却发现鼠标的点击、拖动、放开也分别能够被当做触摸行为。

例中 listener 的 onTouchBegan、onTouchMoved、onTouchEnded 三个成员函数必须显式

地定义，此处使用了 lambda 函数。以下称这三个回调函数为响应函数，因为它们的意义为“当某件事情发生时，应当执行什么判断或操作”。onTouchBegan 即当触摸开始时所做的事情；onTouchMoved 即当触摸已经开始并且移动了的时候所做的事情；onTouchEnded 即当触摸已经开始（可能移动过）并且此时放开了，所要做的事情。

onTouchBegan 只接受 bool 返回值的函数，而另外两个则只接受 void 返回值的函数。只有当 onTouchBegan 函数返回了 true 的时候，后面两个函数才会“开启”，并且当 onTouchEnded 函数生效后，隐式地“关闭”后两个函数。

-对 Touch *t 与 Event *e 的充分发挥-

作为 Touch 指针类型的实体，t 有以下常用功能可用：

auto p = t->getLocation();//返回当前触摸点的位置，类型为 Vec2（cocos 库似乎将 Vec2 当做课程 GUI 库中的 Point 来用）

作为 Event 指针类型的实体，e 有以下常用功能可用：

auto s = e->getCurrentTarget();/*返回当前与之联系的对象，其中这层联系是由 Director::getInstance()->getEventDispatcher()->addEventListenerWithSceneGraphPriority(listener, movable);语句确定的，所以 s 的类型不定，但大体上来说是相应的指针类型，指向的是与之关联的对象，所以要调用成员函数的话需要用->符号。*/

一个较为明显的问题是，Vec2 的名称中就已经显示，它是二元的，即它可能没法支持三维。三维情况下可能会有三维的点类型来支持监听器，又或者按照投屏影像仍然使用二维的点。由于我们已经不打算在这一项目中实现 3D 棋盘，关于 3D 的功能我们也就没有充分寻找研究。

3.1.9 回调函数的两个例子

监听器回调函数书写范例：

```
listener->onTouchBegan = [](Touch * t, Event * e) {
        if  (e->getCurrentTarget()->getBoundingBox().containsPoint(t->getLocation())) {
            return true;
        }
        return false;
    };
listener->onTouchEnded = [fixed1, fixed2](Touch * t, Event * e) {
      auto s = e->getCurrentTarget();//这是什么？
      auto p = t->getLocation();
      if (fixed1->getBoundingBox().containsPoint(p)) {
          auto to = MoveTo::create(0.01, fixed1->getPosition());
          s->runAction(to);
      }
      if (fixed2->getBoundingBox().containsPoint(p)) {
          auto to = MoveTo::create(0.01, fixed2->getPosition());
          s->runAction(to);
      }
      auto to = MoveTo::create(0.01, p);
```

```
        s->runAction(to);
    }
```

监听器的回调函数传入的为一个 Touch 类型的指针和一个 Event 类型的指针。

一个很好的习惯是，在定义实体的时候就顺便用 lambda 函数定义回调函数，例如：

```
    auto item = MenuItemFont::create("Test Scene", [&](Ref * sender) {
            Director::getInstance()->pushScene(TestScene::create());
            });
```

这样能显得简洁得多，不过也少了复用的机会。

3.1.10 关于 addChild 函数

```
this->addChild(item,number);
```

作为节点的子类型，Scene 是可以添加与之关联的子节点的，方法是调用 addChild 函数。item 为节点 Node 类型，一般为 Label、Sprite、Menu 等类型。number 是 int 类型，代表优先级，该数越大优先级越高（越处在表面，显示时会覆盖优先级低的图形）。如果同属于一个优先级，则后来居上。

3.1.11 关于 xxx::create 函数

目前见到过的 create 函数有：

```
auto sprite = Sprite::create("sprite.png");
                        //图片
auto label = Label::createWithTTF("some text here", "fonts/Marker Felt.ttf", 24);
                            // 显示文字          字体                    字符大小
auto menuitemimage = MenuItemImage::create(
                                    "CloseNormal.png",//平常状态下显示的图片
                                    "CloseSelected.png",//选中状态下显示的图片。“选中”指的是鼠标在其范围内按下（左键）。
                                    CC_CALLBACK_1(HelloWorld::menuCloseCallback, this)/*回调函数*/);
auto menu1 = Menu::create(menuitemimage, NULL);
auto menuitemfont = MenuItemFont::create("some text here", CC_CALLBACK_1(HelloWorld::menuCloseCallback, this));
auto menu2 = Menu::create(menuitemfont, NULL);
auto menu3 = Menu::createWithArray(menuitems);//menuitems 的类型为 vector<MenuItem*>
auto listener = EventListenerTouchOneByOne::create();
```

Cocos 库给出了预定义函数 CREATE_FUNC，其代码细节如下：

```
#define CREATE_FUNC(__TYPE__) \
static __TYPE__* create() \
{ \
    __TYPE__ *pRet = new(std::nothrow) __TYPE__(); \
    if (pRet && pRet->init()) \
    { \
        pRet->autorelease(); \
```

```
        return pRet; \
    } \
    else \
    { \
        delete pRet; \
        pRet = nullptr; \
        return nullptr; \
    } \
}
```

CREATE_FUNC 较为方便地定义了空间管理的创造函数，在本游戏的编写中也较常使用。

3.1.12 定时器

```
schedule(schedule_selector(HelloWorld::updateCustom),1.0f, kRepeatForever,0);
```

schedule 是场景类的成员函数，可以做到反复执行某个函数，时间间隔可自定义。这对于实现 AI 自动走棋是非常重要的。

3.1.13 更改窗口大小和分辨率

更改窗口大小：

```
auto director = Director::getInstance();
auto glview = director->getOpenGLView();
glview->setFrameSize(480,320);
//或者啰嗦一点
Director::getInstance()->getOpenGLView()->setFrameSize(480, 320);
```

更改窗口分辨率：

```
auto director = Director::getInstance();
auto glview = director->getOpenGLView();
glview->setDesignResolutionSize(1280,720,ResolutionPolicy::SHOW_ALL);
//或者啰嗦一点
Director::getInstance()->getOpenGLView()->setDesignResolutionSize(1280,720,ResolutionPolicy::
SHOW_ALL);
```

注意，这两者是单独进行的，也就是说最好窗口大小和分辨率一同设置。

3.1.14 场景转换

方式一：push 与 pop

```
Director::getInstance()->pushScene(SomeScene::create());
Director::getInstance()->popScene();
```

方式二：直接转换

```
Director::getInstance()->replaceScene(SomeScene::create());
```

3.1.15 退出游戏

```
Director::getInstance()->end();
```

3.1.16 其他

作为一款非常火热的游戏开发库，Cocos 还有许多其他功能待开发，本次游戏开发由于时间精力有限，没有非常深入地研究。

3.2 类设计

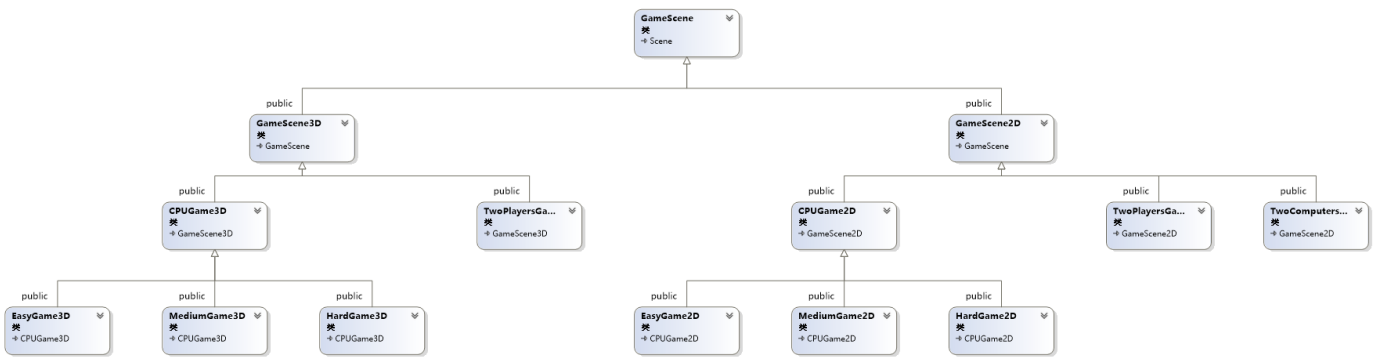
类设计主要分为三个部分，一部分是场景类，一部分是棋盘类前端，一部分是棋盘类后端。棋盘类是中途代码重构的时候才分成了前后两端，在此处便不展示重构前的代码，只展示最终确定的类设计。

3.2.1 场景类

代码集中在 GameScenes.h 中。

```
4   #include "cocos2d.h"
5   #include<string>
6   #include<vector>
7   #include "GameWidget.h"
8   #include "GameAI.h"
9
10  using std::string;
11  using std::vector;
12  USING_NS_CC;
13
14  class GameScene { ... };
55
56  class GameScene2D { ... };
88
89  class GameScene3D { ... };
101
102 class CPUGame2D { ... };
113
114 class CPUGame3D { ... };
124
125 class TwoPlayersGame2D { ... };
135
136 class TwoComputersGame2D { ... };
152
153 class TwoPlayersGame3D { ... };
163
164 class EasyGame2D { ... };
171
172 class MediumGame2D { ... };
179
180 class HardGame2D { ... };
187
188 class EasyGame3D { ... };
194
195 class MediumGame3D { ... };
201
202 class HardGame3D { ... };
208
209
210 #endif
```

类之间关系见类图：



此处重点说明一下几个虚函数是如何实现多态的。

```
14  class GameScene : public cocos2d::Scene
15  {
16  public:
17      // ...
19      vector<Label*> vlabel_PlayingPlayer;
20      vector<Sprite*> vGameOverPlate;
21      vector<Label*> vGameOverLabel;//[0]:LabelGameOver [1]:LabelWhoWin [2]: LabelStartNew
22      vector<Sprite*> vPawnPromotionSprites;//[0]:Plate [1]:Pawn [2]:Rook [3]:Knight [4]:Bishop [5]:Queen
23      vector<Label*> vPawnPromotionLabel;
24      bool p1_is_white = true;//white is offensive
25      bool game_is_over = false;
26      bool menuShowing = false;
27      bool promotionPlateShowing = false;
28      //FUNCTIONS:
29      static cocos2d::Scene* createScene();
30
31      virtual bool init();
32
33      void initMainMenu();
34      virtual void initTestButtons() {};//test
35      virtual void initGameInfo() {};
36      virtual void initGameRange() {};
37      virtual void refreshGameRange() {};
38      virtual void initPlayingPlayerInfo();
39      virtual void refreshPlayingPlayerInfo() {};
40      virtual void initButtons() {};//悔棋、投降等
41      virtual void initGameAI() {};
42      virtual void initListener() {};
43      virtual void initGameOverInfo() {};
44      virtual void initPawnPromotionInfo() {};
45      virtual void initOther() {};
46      virtual string player_white() { return string(""); }
47      virtual string player_black() { return string(""); }
48
49      // ...
51
52      // implement the "static create()" method manually
53      CREATE_FUNC(GameScene);
54  };
```

GameScene 类型定义了数个虚成员函数，这些成员函数大多在其子类中被覆盖。例如 initGameInfo()函数，其作用是在左上角显示出当前棋局的维度和难度，因此不同的场景子类必然会有不同的实现。

initPlayingPlayerInfo()和 refreshPlayingPlayerInfo()函数的作用是创建和刷新当前应采取行动的一方的信息。

更为重要的是 initGameAI()、initListener()两个函数，顾名思义，他们掌管的是人类与机器的行为响应。例如在简单难度的 2D 场景类型中，有：

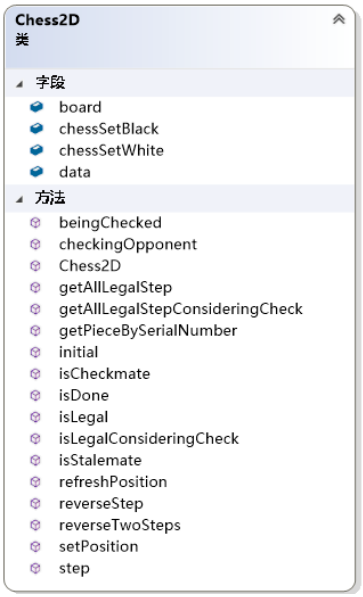
```
164  class EasyGame2D :public CPUGame2D
165  {
166  public:
167      virtual void initGameInfo()override;
168      virtual void initGameAI() { best_step = &random_AI_step; };
169      CREATE_FUNC(EasyGame2D);
170  };
```

而在困难难度的 2D 场景类型中，有：

```
180  class HardGame2D :public CPUGame2D
181  {
182  public:
183      virtual void initGameInfo()override;
184      virtual void initGameAI() { best_step = &alpha_beta_AI_step; };
185      CREATE_FUNC(HardGame2D);
186  };
```

这样他们的父类中调用过 initGameAI()的函数就会按照不同的方式来进行初始化人工智能算法的函数指针。

3.2.2 棋盘类前端



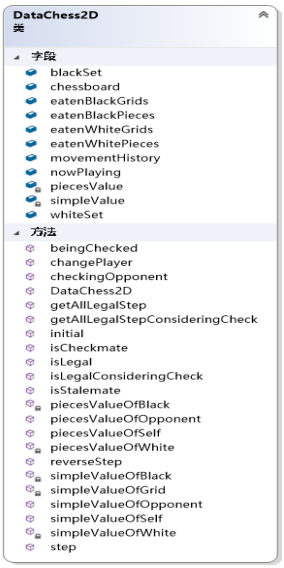
前端比较简洁，聚敛到一个单独的 Chess2D 类来实现。

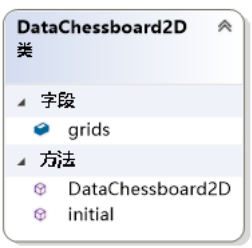
配合后端类型，前端的 step()、reverseStep()直接调用成员 data 的相应方法，并且还单独书写了处理图形变化的代码。

前端的 isDone()显示了它当前是否有正在移动的棋子，如果有，返回 false，否则返回 true。这样就不会出现机器抢步的情况。由于 Sprite 类型的移动是用矢量叠加来实现的，因此如果某一个棋子既被移动，又在移动还未结束的时候被吃掉，它的终点就可能不受控制。

更多方法细节详见代码。一般来说看方法的标识符就能知道它的意义和用法。

3.2.3 棋盘类后端





这是三个比较大的类型，所表示的意义分别是整个棋盘的信息、某一方的全部棋子、棋盘。其中后两个基本上只是包裹类型，即将多个同种类型的实体进行合并处理，而真正复杂的是 DataChess 这个详细数据类型。它定义的成员函数中，有这几个是最为重要的：

isLegal()：
判断当前着法是否合乎一般规则（不包括应将规则）。

isLegalConsideringCheck()：
判断当前着法是否合乎应将规则（当然同时也需要合乎一般规则）。

step()：
按照传入的着法走一步，移动棋子位置、更改棋子是否已经走动、是否已经被吃掉、以及升变信息，并且将这一步记录到历史中。

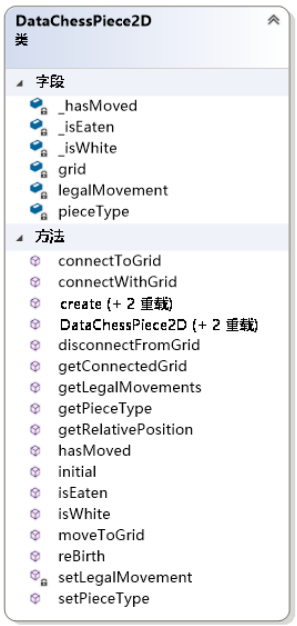
reverseStep()：
根据历史记录的最后一步，还原上一步的棋局。

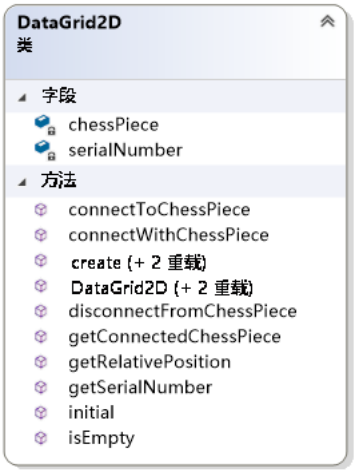
getAllLegalStep()：
得到所有能走的合法着法。

getAllLegalStepConsideringCheck()：
得到所有能走的合法着法，考虑应将规则。

isCheckmate()：
判断当前应当走子的一方是否已经被将死。

isStalemate()：
判断当前走子的一方是否已经被逼和。





这两个类型是封装起来的小类型，其成员类型都为私有，只能通过接口对它们进行更改。

DataChessPiece2D 的成员中，_hasMoved 表示它是否已经移动过，这将影响兵是否可以向前走两格；_isEaten 表示它是否已经被吃了，这将影响棋子是否可以贡献着法；_isWhite

表示它是否是白子；grid 指向一个 DataGrid2D 类型实体，这是这个棋子附着着的格点；legalMovement 是一个函数指针，调用它可以获得该棋子当前可走的所有着法（不考虑应将规则）；pieceType 表示它的类型。

connectToGrid 方法让棋子附着在某格点上，但这将会有一个对应性的问题：如果两个棋子同时对应一个格点，将会带来逻辑错误。

connectWithGrid 方法让棋子附着在某格点上，同时开除这个格点上原有的棋子。这也会带来对应性问题。

实际使用中，一个 DataChessSet2D 类型中所有的棋子都会找到属于他们自己的格点，即便是被吃掉的棋子也会找到棋盘外的一个位置放下，这样对应性的问题就被解决了。

getRelativePosition 方法按照棋子附着着的格点的序列号（之后介绍），返回它在屏幕上应有的相对坐标（相对屏幕中央），这将为前端提供便利。

DataGrid2D 的成员中，chessPiece 表示附着在它上面的棋子，这个值可以为 nullptr（而且经常为 nullptr）；serialNumber 表示它的序列号，如果它在棋盘中，那么左上角的格子序列号为 0，右上角的格子序列号为 7，左下角的格子序列号为 56，右下角的格子序列号为 63，如果它不在棋盘中，那么给左侧黑方被吃掉的子准备的格子的序列号为-1~-16，给右侧白方被吃掉的子准备的格子的序列号为-17~-32。

isEmpty 方法返回这个格子是否有棋子附着的信息。

getRelativePotision 方法按照序列号，返回它在屏幕上应有的相对坐标（相对屏幕中央），这将为前端提供便利。

3.3 场景转换的具体实现

开始场景中：

```
46      auto font_item_new_game = MenuItemFont::create("New Game", [&](Ref* sender) {
47          Director::getInstance()->pushScene(DimensionSelectScene::create());
48          });
```

实现了开始界面到维度选择界面的转换。

维度选择场景中：

```
44      auto font_item_2D = MenuItemFont::create("2D", [&](Ref* sender) {
45          Director::getInstance()->pushScene(DifficultySelectScene::createScene(2));
46          });

51      auto font_item_3D = MenuItemFont::create("3D", [&](Ref* sender) {
52          Director::getInstance()->pushScene(DifficultySelectScene::createScene(3));
53          });
```

实现了维度选择场景到难度选择场景的转换。

```
58      auto font_item_back = MenuItemFont::create("Back", [&](Ref* sender) {
59          Director::getInstance()->popScene();
60          });
```

实现了退出维度选择场景。

难度选择场景中：

```
46      auto font_item_easy = MenuItemFont::create("Easy", [&](Ref* sender) {
47          Director::getInstance()->popScene();
48          if (dimension == 2) Director::getInstance()->replaceScene(EasyGame2D::create());
49          else Director::getInstance()->replaceScene(EasyGame3D::create());
50          });

56      auto font_item_medium = MenuItemFont::create("Medium", [&](Ref* sender) {
57          Director::getInstance()->popScene();
58          if (dimension == 2) Director::getInstance()->replaceScene(MediumGame2D::create());
59          else Director::getInstance()->replaceScene(MediumGame3D::create());
60          });

66      auto font_item_hard = MenuItemFont::create("Hard", [&](Ref* sender) {
67          Director::getInstance()->popScene();
68          if (dimension == 2) Director::getInstance()->replaceScene(HardGame2D::create());
69          else Director::getInstance()->replaceScene(HardGame3D::create());
70          });

76      auto font_item_2p = MenuItemFont::create("2 Players", [&](Ref* sender) {
77          Director::getInstance()->popScene();
78          if (dimension == 2) Director::getInstance()->replaceScene(TwoPlayersGame2D::create());
79          else Director::getInstance()->replaceScene(TwoPlayersGame3D::create());
80          });

86      auto font_item_2c = MenuItemFont::create("2 Computers", [&](Ref* sender) {
87          if (dimension == 2) {
88              Director::getInstance()->popScene();
89              Director::getInstance()->replaceScene(TwoComputersGame2D::create());
90          }
91          });

97      auto font_item_back = MenuItemFont::create("Back", [&](Ref* sender) {
98          Director::getInstance()->popScene();
99          });
```

实现了难度选择场景到各游戏场景的转换，以及退回到开始界面的功能。

3.4 棋盘可视化的具体实现

主要函数为 Chess2D 类型的成员函数 initial()和 refreshPosition()。

initial()方法初始化所有成员，包括各个棋子和棋局数据，而可视化的棋子则按照棋子最初的摆法确定其位置。

refreshPosition()方法刷新可视化棋子的位置，这样在类型为 DataChess2D 的数据走完一步之后，可视化棋子就能够移动到它们该去的位置。

3.5 用户行为响应的具体实现

用户行为响应主要定义在 GameScene2D::initListener()中。

```
195     from_click->onTouchBegan = [this,middle](Touch* t, Event* e) {
196         if (game_is_over)return false;
197         if (chosen)return false;
198         if (chosen_index != -1)return false;
199         if (menuShowing)return false;
200         if (promotionPlateShowing)return false;
201         auto s = e->getCurrentTarget();
202         auto p = t->getLocation();
203         if (s->getBoundingBox().containsPoint(p)) {
204             if (positionToIndex(p, middle) != -1) {
205                 chosen_index = positionToIndex(p, middle);
206                 return true;
207             }
208             else return false;
209         }
210         return false;
211     };
212     from_click->onTouchEnded = [this,middle](Touch* t, Event* e) {
213         if (chosen)return;
214         float up = 15;
215         auto s = e->getCurrentTarget();
216         auto p = t->getLocation();
217         index i = positionToIndex(p, middle);
218         if (i == -1) {
219             chosen = false;
220             chosen_index = -1;
221             return;
222         }
223         if (s->getBoundingBox().containsPoint(p)) {
224             if (i == chosen_index ) {
225                 if (chess.data.nowPlaying == Player::white && (player_white() != "Computer")) {
226                     if (chess.data.chessboard.grids[i]->isEmpty()) {
227                         chosen = false;
228                         chosen_index = -1;
229                         return;
230                     }
231                     auto chosen_piece = chess.data.chessboard.grids[i]->getConnectedChessPiece();
232                     if (!chosen_piece->isWhite()) {
233                         chosen = false;
234                         chosen_index = -1;
235                         return;
236                     }
237                     //if efficient
238                     auto chosen_sprite_piece = chess.getPieceBySerialNumber(i);
239                     if (chosen_sprite_piece != nullptr) {
240                         chess.getPieceBySerialNumber(i)->runAction(MoveBy::create(0.1, Vec2(0, up)));
241                         chosen = true;
242                     }
243                     return;
244                     //
245                 }
246                 else if (chess.data.nowPlaying == Player::black && (player_black() != "Computer")) {
247                     if (chess.data.chessboard.grids[i]->isEmpty()) {
248                         chosen = false;
249                         chosen_index = -1;
250                         return;
251                     }
252                     auto chosen_piece = chess.data.chessboard.grids[i]->getConnectedChessPiece();
253                     if (chosen_piece->isWhite()) {
254                         chosen = false;
255                         chosen_index = -1;
256                         return;
257                     }
258                     //if efficient
259                     chess.getPieceBySerialNumber(i)->runAction(MoveBy::create(0.1, Vec2(0, up)));
260                     chosen = true;
261                     return;
262                     //
263                 }
264                 chosen = false;
265                 chosen_index = -1;
266                 return;
267             }
268             else {
269                 chosen = false;
270                 chosen_index = -1;
271                 return;
272             }
273         }
274     };
```

```
to_click->onTouchBegan = [this,middle](Touch* t, Event* e) {
    if (game_is_over)return false;
    if (!chosen)return false;
    if (chosen_index == -1)return false;
    if (menuShowing)return false;
    if (promotionPlateShowing)return false;
    auto p = t->getLocation();
    index i = positionToIndex(p, middle);
    chosen_index2 = i;
    return true;
};
to_click->onTouchEnded = [this,middle](Touch* t, Event* e) {
    if (!chosen)return;
    if (chosen_index == -1)return;
    float up = 15;
    auto s = e->getCurrentTarget();
    auto p = t->getLocation();
    index i = positionToIndex(p, middle);
    auto chosen_piece = chess.data.chessboard.grids[chosen_index]->getConnectedChessPiece();
    if (chosen_index2 == -1) {
        chess.getPieceBySerialNumber(chosen_index)->runAction(MoveBy::create(0.1, Vec2(0, -up)));
    }
    else {
        auto movement=ChessPieceMovement(chosen_piece->getPieceType(), chosen_index, chosen_index2,
            (chess.data.chessboard.grids[chosen_index2]->isEmpty() ? PieceType::NONE : chess.data.chessboard.grids[chosen_index2]->getConnectedChessPiece()->getPieceType()),
            !chosen_piece->hasMoved());
        if (chess.isLegal(movement)) {
            //if Effecient
            if (movement.pieceType == PieceType::Pawn && ((movement.to / 8 == 0) || (movement.to / 8 == 7))) {//Pawn's promotion
                chess.data.movementHistory.push_back(movement);
                PawnPromotionInfoDragDown();
            }
            else {
                chess.step(movement);
                refreshPlayingPlayerInfo();
                if (movement.eatType == PieceType::King) {
                    game_is_over = true;
                    string winner;
                    if (chess.data.nowPlaying == Player::black) {
                        winner += "White (";
                        winner += player_white();
                        winner += ")";
                    }
                    else {
                        winner += "Black (";
                        winner += player_black();
                        winner += ")";
                    }
                    winner += " Wins !";
                    GameOverInfoRiseUpWithString(winner);
                }
            }
        }
        else {
            chess.getPieceBySerialNumber(chosen_index)->runAction(MoveBy::create(0.1, Vec2(0, -up)));
        }
    }
    chosen = false;
    chosen_index = -1;
    chosen_index2 = -1;
};
```

基本逻辑是，记录用户两次点击的首尾序列号信息，根据两者生成走法，如果走法合法就执行，如果走法不合法就不执行。

3.6 电脑自动随机走棋的具体实现

主要实现在：

```
class EasyGame2D :public CPUGame2D
{
public:
    virtual void initGameInfo()override;
    virtual void initGameAI()  { best_step = &random_AI_step; };
    CREATE_FUNC(EasyGame2D);
};
```

```
ChessPieceMovement random_AI_step(DataChess2D* ch) {
    //Artificial Idiot
    vector<ChessPieceMovement> collect = ch->getAllLegalStepConsideringCheck();
    if (collect.size() == 0) {
        return ChessPieceMovement();
    }
    auto i = U(E) % collect.size();
    return collect[i];
}
```

后者在 GameAI.cpp 文件中。基本思路是，获取当前可走的所有合法走法，并且需要考虑应将规则。一个较为明显的问题是，该函数必定需要返回一个走法，而事实上在将死和逼和的情况下这一方是无棋可走的。也就是说，要避免返回一个空走法的情况，就要在别的地

方预先判断当前棋局是否是将死、逼和的状况。

3.7 应将和终局判定的具体实现

应将规则：

```
691    bool DataChess2D::checkingOpponent()
692    {
693        vector<ChessPieceMovement>temp_move_all = getAllLegalStep();
694        for (auto m : temp_move_all) {
695            if (m.eatType == King)return true;
696        }
697        return false;
698    }
699    bool DataChess2D::beingChecked() {
700        changePlayer();
701        if (checkingOpponent()) {
702            changePlayer();
703            return true;
704        }
705        changePlayer();
706        return false;
707    }
```

当己方所有走法中，存在一个能够吃到对方王的走法，那么当前情况下就是我方 check 对方。反之，站在对方的角度，如果找到了一个能够吃到我方王的走法，那么当前情况下就是我方被 check。

终局判定：

```
731    bool DataChess2D::isCheckmate()
732    {
733        auto collect = getAllLegalStepConsideringCheck();
734        if (collect.size() == 0 && beingChecked())return true;
735        return false;
736    }
737    bool DataChess2D::isStalemate()
738    {
739        auto collect = getAllLegalStepConsideringCheck();
740        if (collect.size() == 0 && !beingChecked())return true;
741        return false;
742    }
```

如果搜索当前考虑应将规则的所有合法走法，为空，那么就必然到达终局判定。如果此时应走方的王正被将军，那么就是将死情形，为 checkmate。如果此时应走方的王没有被将军，但也无子可走了，那么就是逼和情形，为 stalemate。

3.8 电脑贪吃走法的具体实现

主要实现在：

```
172     class MediumGame2D :public CPUGame2D
173     {
174     public:
175         virtual void initGameInfo()override;
176         virtual void initGameAI() { best_step = &gluttonous_AI_step; };
177         CREATE_FUNC(MediumGame2D);
178     };
```

```
22    //只看一步，找到最佳的走法
23  ChessPieceMovement gluttonous_AI_step(DataChess2D* ch) {
24      vector<ChessPieceMovement> collect = ch->getAllLegalStepConsideringCheck();
25      if (collect.size() == 0) {
26          return ChessPieceMovement();
27      }
28      ChessPieceMovement best_eat_step = collect[0];//能吃到最厉害的子的走法
29      ChessPieceMovement best_value_step = collect[0];//不能吃子，但走后局势评价函数最大的走法
30      double prescore{ 0 }, score{ 0 };//不能吃子时，走后的局势改变量
31      for (ChessPieceMovement m : collect) {
32          if (m.eatType > best_eat_step.eatType)//若当前走法能吃到的子更好
33              best_eat_step = m;
34          else {          //若当前走法不能吃到子
35              switch (m.pieceType) { //现在默认电脑方为黑方
36              case PieceType::Pawn:
37                  score = pawn_value_black[m.from] - pawn_value_black[m.to];
38                  break;
39              case PieceType::Rook:
40                  score = rook_value_black[m.from] - rook_value_black[m.to];
41                  break;
42              case PieceType::Knight:
43                  score = knight_value_black[m.from] - knight_value_black[m.to];
44                  break;
45              case PieceType::Bishop:
46                  score = bishop_value_black[m.from] - bishop_value_black[m.to];
47                  break;
48              case PieceType::Queen:
49                  score = queen_value_black[m.from] - queen_value_black[m.to];
50                  break;
51              case PieceType::King:
52                  score = king_value_black[m.from] - king_value_black[m.to];
53                  break;
54              default:
55                  score = 0;
56              }
57              if (prescore <= score) {
58                  best_value_step = m;
59                  prescore = score;
60              }
61          }
62      }
63      if (best_eat_step.eatType != PieceType::NONE) {
64          return best_eat_step;
65      }
66      else {
67          return best_value_step;
68          //auto i = U(E) % collect.size();
69          //return collect[i];
70      }
71  }
```

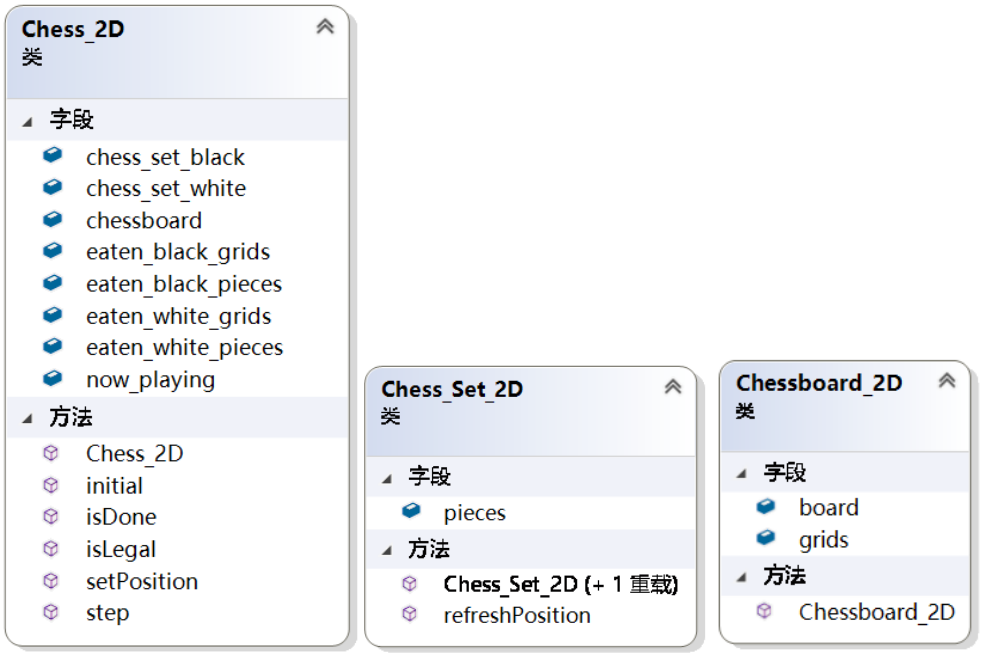
其中 gluttonous 意为“暴食”、“饕餮”，这一 AI 逻辑所遵循的正是“见子就吃”的原则。如果有子可吃，则不计后果地吃掉它。如果无子可吃，则在现有走法中选取一个最有价值的走法。其中这个价值是用 Value.h 中的矩阵存储的。

3.9 前后端分离的代码重构

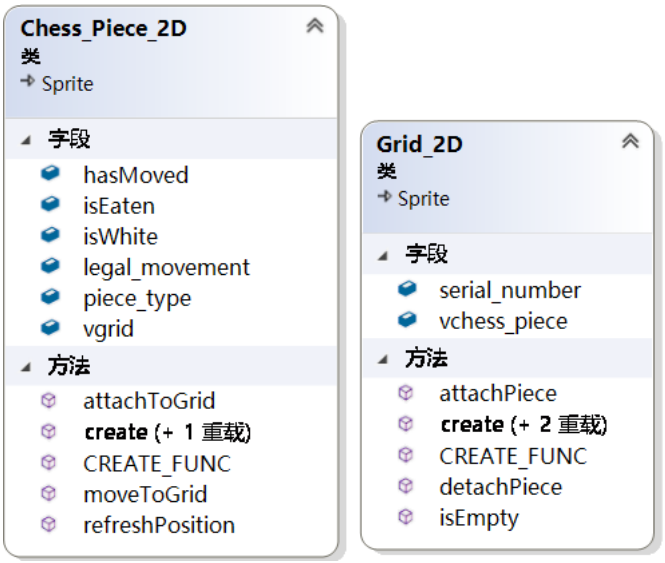
在实现 AI 算法的过程中，我们发现即便是简单的 Random 走法，如果不进行前后端分离处理，用起来也是十分麻烦。加之之后需要设计深度搜索的走法，我们认为前后端分离的

代码重构势在必行。

重构前：



这三者主要为包含类型，一个 Chess_Set_2D 的实体将包含某一方的全部棋子，一个 Chessboard_2D 类型包含全部格点和一个 sprite 类型的棋盘。



这两个类型继承自 Sprite，具有精灵的可视化功能，设计思路基本上也和重构之后的一样。

但是在使用过程中发现，这样的设计有如下问题：

①Grid_2D 类型并没有包含任何图片，而是以无图片的形式出现的，这样还不如不让它继承自 Sprite，而让它继承自 Node。

②前后端不分离，那么在进行 alpha-beta 剪枝搜索的时候（实际上是任何有深度搜索），就难以在保持游戏窗口内容不变的情况下进行推演。并且，推演的时候拖拽着大量而无用的多余内容（指的是只在 Sprite 中用于可视化，而对棋局并没有本质影响的内容），也会极大程度上降低效率。

因此，我们对这部分类设计代码进行了重构，而尽可能不改变其接口。

重构之后的类设计已经在之前展示，在此便不赘述。

3.10 悔棋按钮的具体实现

考虑到双人对战时悔棋往往只悔一步，而人机对战时悔棋必须悔两步（否则电脑会自作主张地又着一步），两者是分开定义的。

双人对战：

```
644  void TwoPlayersGame2D::initButtons()
645  {
646      auto visibleSize = Director::getInstance()->getVisibleSize();
647      auto origin = Director::getInstance()->getVisibleOrigin();
648      auto regretButton = Sprite::create("LongButton.png");
649      regretButton->setPosition(Vec2(origin.x + visibleSize.width - regretButton->getContentSize().width / 2, origin.y + regretButton->getContentSize().height / 2));
650      auto regretLabel = Label::createWithTTF("Regret", "fonts/Marker Felt.ttf", 24);
651      regretLabel->setPosition(regretButton->getPosition());
652      auto regretListener = EventListenerTouchOneByOne::create();
653      regretListener->onTouchBegan = [this](Touch* t, Event* e) {
654          if (menuShowing)return false;
655          if (promotionPlateShowing)return false;
656          if (!chess.isDone())return false;
657          if (e->getCurrentTarget()->getBoundingBox().containsPoint(t->getLocation())) return true;
658          return false;
659      };
660      regretListener->onTouchEnded = [this](Touch* t, Event* e) {
661          if (e->getCurrentTarget()->getBoundingBox().containsPoint(t->getLocation())) {
662              chess.reverseStep();
663              refreshPlayingPlayerInfo();
664          }
665      };
666
667      this->addChild(regretButton);
668      this->addChild(regretLabel);
669      Director::getInstance()->getEventDispatcher()->addEventListenerWithSceneGraphPriority(regretListener, regretButton);
670  }
```

人机对战：

```
592  void CPUGame2D::initButtons()
593  {
594      auto visibleSize = Director::getInstance()->getVisibleSize();
595      auto origin = Director::getInstance()->getVisibleOrigin();
596      auto regretButton = Sprite::create("LongButton.png");
597      regretButton->setPosition(Vec2(origin.x + visibleSize.width - regretButton->getContentSize().width / 2, origin.y + regretButton->getContentSize().height / 2));
598      auto regretLabel = Label::createWithTTF("Regret", "fonts/Marker Felt.ttf", 24);
599      regretLabel->setPosition(regretButton->getPosition());
600      auto regretListener = EventListenerTouchOneByOne::create();
601      regretListener->onTouchBegan = [this](Touch* t, Event* e) {
602          if (menuShowing)return false;
603          if (promotionPlateShowing)return false;
604          if ((chess.data.nowPlaying == white ? player_white() : player_black()) == "Computer")return false;
605          if (!chess.isDone())return false;
606          if (e->getCurrentTarget()->getBoundingBox().containsPoint(t->getLocation())) return true;
607          return false;
608      };
609      regretListener->onTouchEnded = [this](Touch* t, Event* e) {
610          if (e->getCurrentTarget()->getBoundingBox().containsPoint(t->getLocation())) chess.reverseTwoSteps();
611          if (game_is_over) {
612              game_is_over = false;
613              GameOverInfoMoveDown();
614          }
615      };
616
617      this->addChild(regretButton);
618      this->addChild(regretLabel);
619      Director::getInstance()->getEventDispatcher()->addEventListenerWithSceneGraphPriority(regretListener, regretButton);
620  }
```

双机对战另述。

3.11 电脑自行对战的具体实现

主要包括 电脑不依靠鼠标点击屏幕触发走棋 和 两个电脑分别执黑执白对战 两部分。

在之前的实现方案中，人机对战时，必须要用鼠标点击屏幕才能够触发电脑行棋，这对于一个象棋游戏来说并不好，最好是能够做到电脑能够自动行棋。在找到了 cocos 库 schedule 功能后，这个问题得到完美解决。

```
341     schedule([this](float f) {
342         if (game_is_over)return;
343         if (menuShowing)return;
344         if (promotionPlateShowing)return;
345         if (!(chess.isDone() && ((chess.data.nowPlaying == white ? player_white() : player_black()) == "Computer")))return;
346         auto movement = (*best_step)(&chess.data);
347         if (chess.isLegal(movement)) {//if it's two players mode, best_step() will return an illegal movement, and still let human take a step.
348             chess.step(movement);
349             refreshPlayingPlayerInfo();
350             if (movement.eatType == PieceType::King) {
351                 game_is_over = true;
352                 string winner;
353                 if (chess.data.nowPlaying == Player::black) {
354                     winner += "White (";
355                     winner += player_white();
356                     winner += ")";
357                 }
358                 else {
359                     winner += "Black (";
360                     winner += player_black();
361                     winner += ")";
362                 }
363                 winner += " Win !";
364                 GameOverInfoRiseUpWithString(winner);
365             }
366         }
367         }, 0.1f,string("StopAlwaysAI"));
```

电脑会不停检测现在是不是该他走了，仿佛在催促玩家赶紧着子一样。

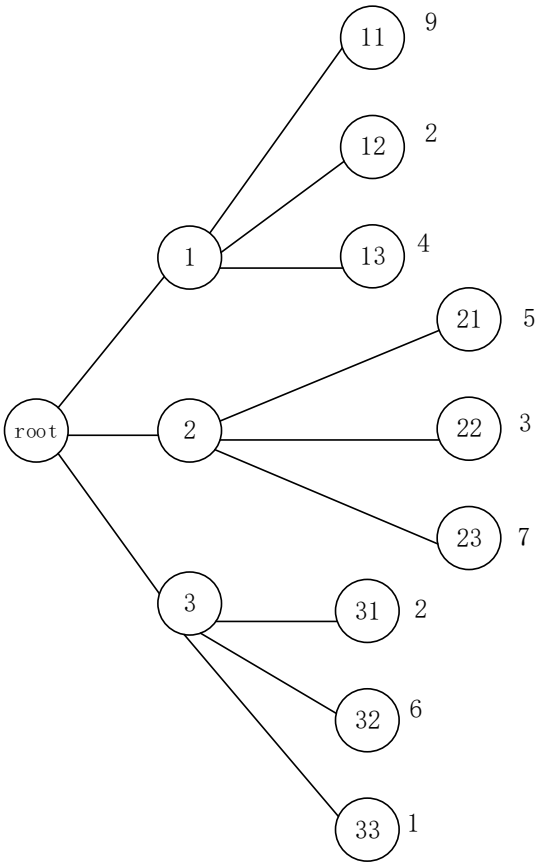
同样，让电脑自己和自己下棋的场景也不难做。

```
755 void TwoComputersGame2D::initListener()
756 {
757     schedule([this](float f) {
758         if (!flag)return;
759         if (game_is_over)return;
760         if (menuShowing)return;
761         if (paused)return;
762         if (!chess.isDone())return;
763         ChessPieceMovement movement;
764         if (chess.isCheckmate()) {
765             game_is_over = true;
766             string winner;
767             if (chess.data.nowPlaying == Player::black) {
768                 winner += "White (";
769                 winner += player_white();
770                 winner += ")";
771             }
772             else {
773                 winner += "Black (";
774                 winner += player_black();
775                 winner += ")";
776             }
777             winner += " Win !";
778             GameOverInfoRiseUpWithString(winner);
779         }
780         else if (chess.isStalemate()) {
781             game_is_over = true;
782             GameOverInfoRiseUpWithString("StaleMate!");
783         }
784         else {
785             flag = false;
786             if (chess.data.nowPlaying == white) movement = (*best_step)(&chess.data);
787             else movement = (*best_step2)(&chess.data);
788             chess.step(movement);
789             refreshPlayingPlayerInfo();
790             flag = true;
791         }
792         }, 0.2f, string("StopAlwaysAI"));
793 }
```

3.12 电脑深度搜索走法的具体实现

这是本 Project 难度最高的一部分。

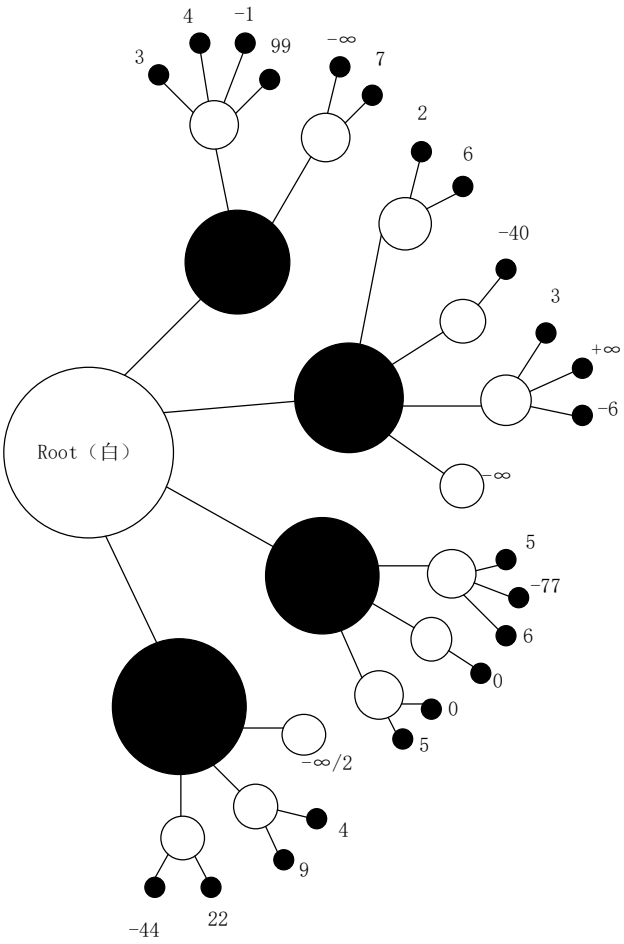
我们所采用的算法与真正意义上的 alpha-beta 算法存在一些差异，准确来说只是一种“零和博弈提前退出式最大最小搜索算法”，但为了好叫所以相关的算法名称仍然叫做 alpha-beta，希望不要搞混。



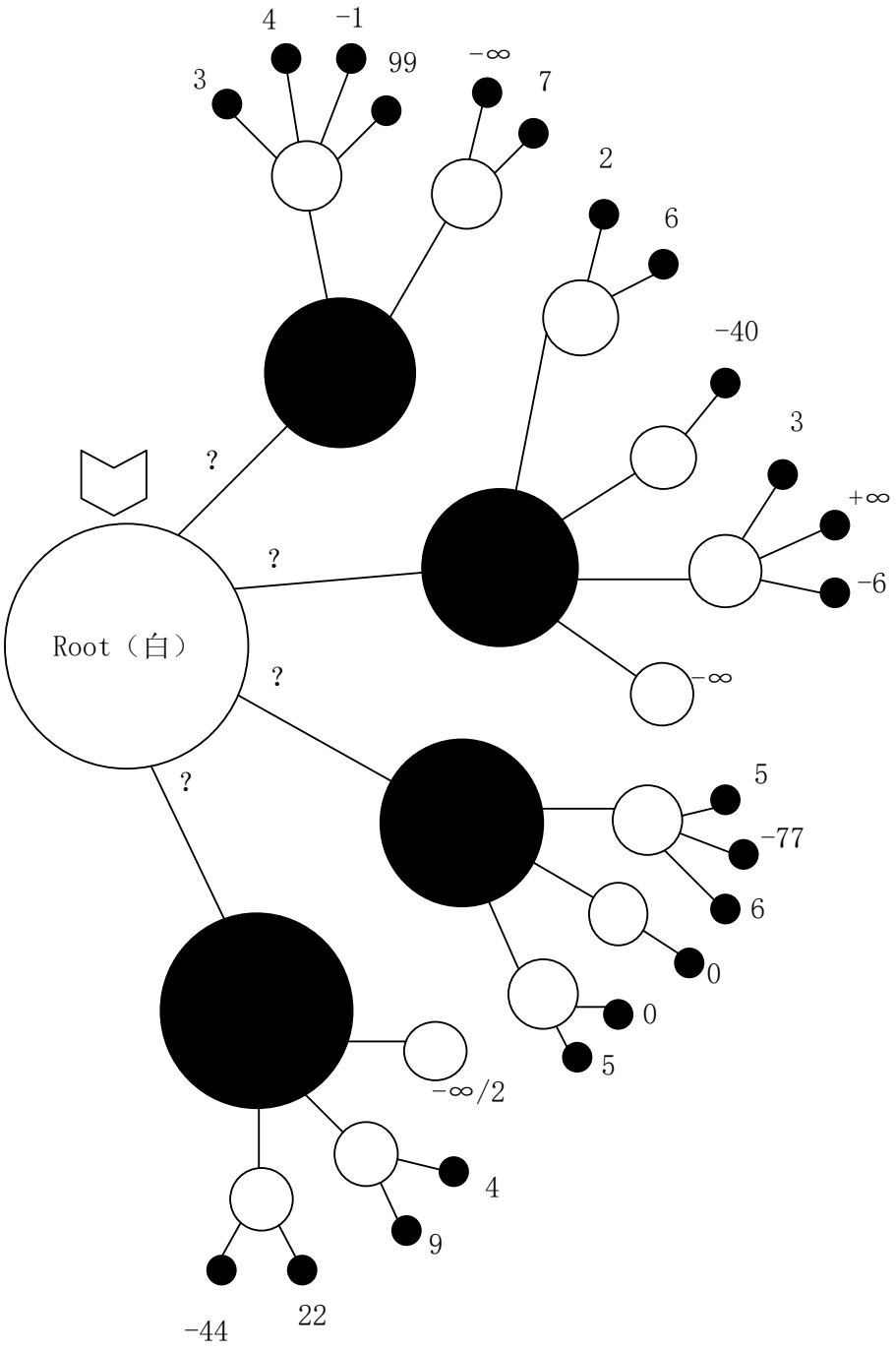
$cd root,maxr=-∞,ceil=+∞
$cd 1,floor=maxr{-∞},min1=+∞
$cd 11,return 9
$cd 1,floor{-∞}<9<min1{+∞},min1=9
$cd 12,return 2
$cd 1,floor{-∞}<2<min1{9},min1=2
$cd 13,return 4
$cd 1,4>min1{2}
return min1{2}
$cd root,min1{2}>maxr{-∞},maxr=min1
$cd 2,floor=maxr{2},min2=+∞
$cd 21,return 5
$cd 2,floor{2}<5<min2{+∞},min2=5
$cd 22,return 3
$cd 2,floor{2}<3<min2{5},min2=3
$cd 23,return 7
$cd 2,7>min2{3}
return min2{3}
$cd root,min2{3}>maxr{2},maxr=min2
$cd 3,floor=maxr{3},min3=+∞
$cd 31,return 2
$cd 3,2<floor{3},return 2
$cd root,2<maxr{3}
return maxr{3}

这是“提前退出最大最小搜索算法”的逻辑。着子方需要从数个情况中挑选，而这些情况又有后续情况，站在对方的角度，为当前着子方挑选一个最不利的情况，就是“从最小里面挑最大”，即最大最小搜索算法。而“提前退出”体现在当搜索进行到节点 31 时，发现当前值小于下限 3，就直接退出，节省了搜索时间。

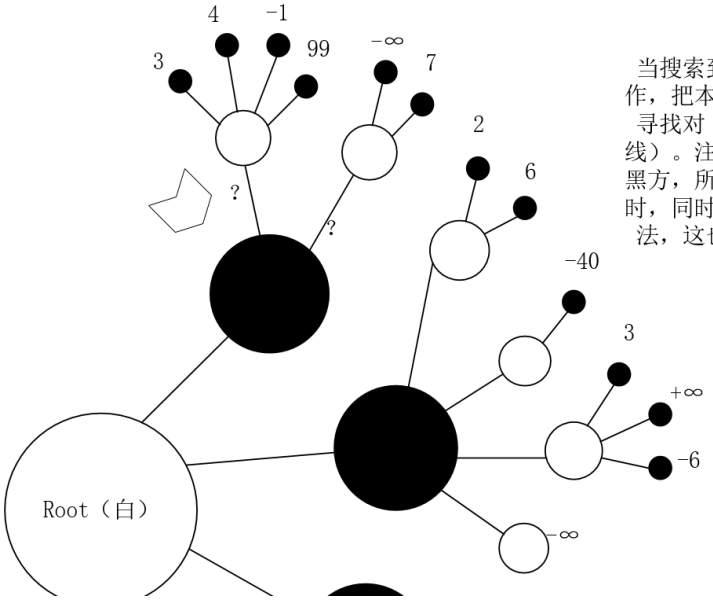
但后来考虑到在本算法中，国际象棋的评分标准为零和博弈，即考察双方的棋力差距，因此没有必要使用一个 floor 和一个 ceil（ceiling）来存储当前的上下限，只需要一个上限 ceil 即可。逻辑如下：



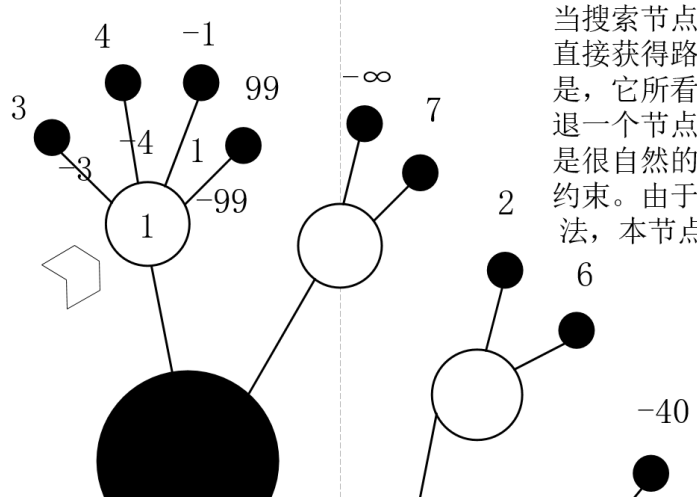
图中数字表示的都是“本方”利益，例如某个黑圆旁显示了当前走子方为黑时，自身的利益。因此如果搜索节点处在其前置节点（轮到白方走）时，白方应对此值取负。之后有具体的例子。



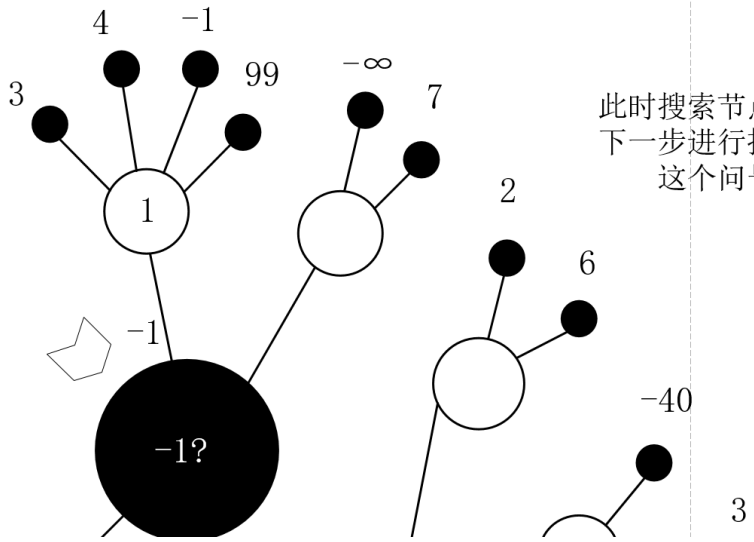
当搜索节点为Root时，程序试图找到连线的受益。



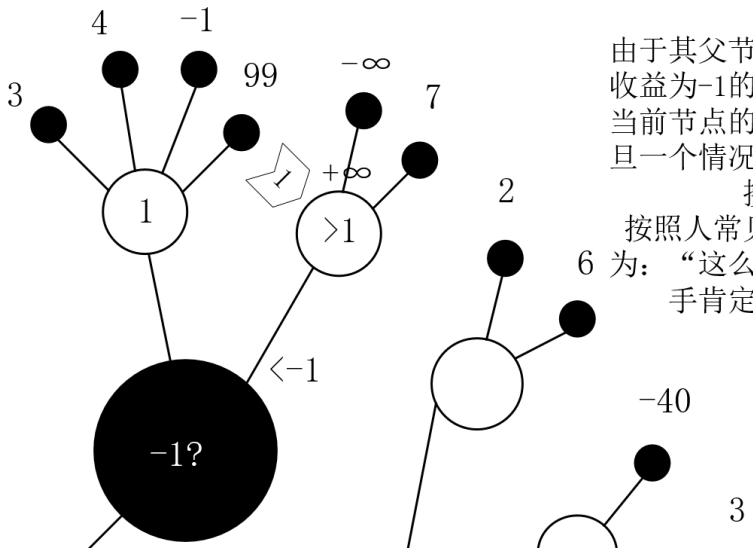
当搜索到图示节点时，重复同样的动作，把本节点当做新的Root节点，仍然寻找对“自己”最有利的着法（即连线）。注意，此处“自己”已经变成了黑方，所以在为自己寻找最有利的着法时，同时就是为白方寻找了最不利的着法，这也是因为有零和博弈的限制。



当搜索节点位于此处时，算法可以直接获得路径的收益。需要注意的是，它所看到的东西是反的。每回退一个节点，收益都要反一遍。这是很自然的，也是因为有零和博弈约束。由于要搜索对自己最好的算法，本节点获得的最大收益就是1



此时搜索节点位于黑色圆圈，下一步进行搜索时，将会带着这个问号进行搜索。



由于其父节点已经获得了一个收益为-1的方案，那么相当于当前节点的天花板就为+1，一旦一个情况优于它，那么就直接退出。

按照人常见的思维可以理解为：“这么走太便宜我了，对手肯定没这么傻。”

由此，只需要搜索算法不断地“为自己考虑”，并且每返回一次都添加一个负号，就只需要一个天花板即可完成整个搜索，并且也享有提前退出的时间优势。

具体代码实现如下。其中 depth 为 0 时就已经有两层搜索。一层是 Root 层向外延伸的部分，这一层需要尽可能多地获取走法，尤其是开局时，任意走一步收益都为零，这样就需要在 20 步中选一步来走。另一层是最末一层，当得到所有走法的时候，就已经可以通过走法的信息来判断这一步是否有价值，因为走法信息中已经包含有什么子吃掉了什么子，以及是否升变成为了其他的子的信息。

```
166 ChessPieceMovement alpha_beta_AI_step(DataChess2D* ch) {
167     if (ch->getAllLegalStepConsideringCheck().size() == 0)return ChessPieceMovement();
168     std::function<int(int depth,int ceil)> ABfunc = [ch,&ABfunc](int depth,int ceil) {
169         const vector<ChessPieceMovement> &all = ch->getAllLegalStepConsideringCheck();
170         if (all.size() == 0)return ch->piecesValueOfOpponent();
171         int now;
172         int max = -INT_MAX;
173         if (depth > 0) {
174             for (int i = 0; i < all.size();i++) {
175                 ch->step(all[i]);
176                 now = ABfunc(depth - 1, -max);
177                 if (now > max) {
178                     max = now;
179                     if (max > ceil) {
180                         ch->reverseStep();
181                         return -max;
182                     }
183                 }
184                 ch->reverseStep();
185             }
186         }
187         else {
188             int v = ch->piecesValueOfSelf();
189             for (int i = 0; i < all.size(); i++) {
190                 now = v + valueOfMovement(all[i]);
191                 if (now > max) {
192                     max = now;
193                     if (max > ceil) {
194                         return -max;
195                     }
196                 }
197             }
198         }
199         return -max;
200     };
```

```
201     vector<ChessPieceMovement> collect;
202     int now;
203     int max = -INT_MAX;
204     auto all = ch->getAllLegalStepConsideringCheck();
205     for (auto m : all) {
206         ch->step(m);
207         if ((ch->eatenBlackPieces + ch->eatenWhitePieces) <= 8) {
208             now = ABfunc(0, -max);
209         }
210         else if((ch->eatenBlackPieces + ch->eatenWhitePieces) <= 24){
211             now = ABfunc(1, -max);
212         }
213         else {
214             now = ABfunc(2, -max);
215         }
216         if (now > max) {
217             max = now;
218             collect.clear();
219             collect.push_back(m);
220         }
221         else if (now == max) {
222             collect.push_back(m);
223         }
224         ch->reverseStep();
225     }
226     auto i = U(E) % collect.size();
227     return collect[i];
228 }
```

3.13 电脑深度搜索走法的优化实现

在一开始的调试过程中，电脑深度搜索走法的速度很慢，深度为零时搜索时间就已经达到了 1.2s，这让人感到必须对它进行一定的优化。优化的方向并不一定是搜索算法本身，而是其它方面。

当时发现 getAllLegalStep 方法非常耗时，查看代码后发现该方法采用的是很没有效率的方式：例如如果是王，那么他会尝试着向上、向下、向左、向右、向斜上、向斜下共八个方向都走一遍，尝试吃掉兵、車、马等六种棋子或不吃，然后依赖 isLegal 来判断是否合法。很明显，如果他试图向上走到一个空格子去吃一个兵，这个走法是非法的，被驳回。以及其他的一些情况，都会被驳回，只留下合法的走法。这样的逻辑是不会有问题的，但是效率极低。因此我决定优化 getAllLegalStep 函数（以及其调用的子函数）。优化后，深度为零时搜索时间降低到 0.6s 左右。

由于保持使用深度为零不能充分发挥算法的效果，在实际操作中采用的是如下策略：如果双方被吃掉的子加起来不超过 8，则深度采用为 0；如果双方被吃掉的子加起来超过 8 但不超过 24，则深度采用为 1；如果双方所剩的子少于 8，则深度采用为 2。

四. 小组分工

1. 项目成员及分工

组长：张朕银 组员：王依婷，易晓玲，陈煜昊，王贵成

分工	人员
开发库的配置与功能了解	陈煜昊，张朕银
搜索图包、制图，图形界面相关代码编写	陈煜昊，张朕银
搜索示例代码	陈煜昊，王贵成
类设计	张朕银
一般 AI 算法	王依婷，易晓玲
剪枝算法	王依婷，易晓玲，张朕银
文档编写	王依婷，易晓玲，张朕银，王贵成

2.项目开发时间表

日期	项目开发进度
5 月 13 日	初定确定棋类游戏类型
5 月 14 日-5 月 20 日	完成项目 SPEC 编写，确定软件设计
5 月 21 日-5 月 27 日	Cocos 图形库相关学习
5 月 28 日-6 月 3 日	Cocos 图形库使用方法基本掌握，游戏界面初步开发
6 月 4 日-6 月 8 日	模块互联与整合，α 测试完成
6 月 8 日-6 月 16 日	完善功能及文档编写

3.项目开发反思

1、对于用户需求以及自身能力评估还需加强，小组因为最初没有准确评估本游戏的开发难度以及用户需求，在项目初期，我们准备实现三维的国际象棋项目，但由于自身编程能力不足，一直没有办法完成三维棋子的建模，卡在了瓶颈之中。最终由于这个问题我们将项目定位为 2D 国际象棋，但在未来的时间里，如果有机会，我们会接着尝试实现 3D 的场景。

2、最初没有明确在开发大的工程时应该如何分工。在后期逐渐的了解和磨合后，我们学会了如何发挥每位成员的长处，成功地完成了项目的开发和后续文件的编纂。

3、编程思想十分重要，要注重编程地规范性，并且应该更深入地理解结构化编程的思想，自顶向下，逐层细化，项目初期由于忽视了设计和规范的重要性，导致不断大规模更改已有的部分，浪费了大量时间。