

现代操作系统 期末项目 实验报告

Billionaire(传奇大富翁)



开发人员

徐伟元 16340261

许倚安 16340265

熊永琦 16340258

目录

- [游戏概述](#)
- [游戏规则](#)
- [游戏运行](#)
- [游戏架构](#)
- [游戏逻辑伪代码设计](#)
- [项目分工](#)
- [开发细节](#)
 - [场景模块\(Scene\)](#)
- [动画模块](#)
 - [控制系统模块](#)
 - [遇到的问题](#)
- [后记/感想](#)

游戏概述

Billionaire(传奇大富翁)是一款线下聚会的单机回合制游戏。其基本游戏规则遵从传统意义上的大富翁游戏，玩家投骰子，前进，买地，买房子，收取过路费等。但本游戏在传统基础上，引入了 **技能** 系统。每回合初始阶段，系统会随机抽取一个技能，玩家可以使用或不使用技能。技能效果也各具特色：增益类，削弱类等。

~~(相信你会和我一样，爱上这款游戏)~~

项目源代码均已开源放在 [github](#)。

游戏规则

4人游戏，游戏右侧边栏会标识当前回合玩家，显示个人信息以及输出事件信息。
游戏主页面左下角有结束回合按钮，掷骰子按钮，右下角有技能按钮，买卖地皮，升级房子按钮

1. 人物动作规则
每回合人物可执行以下动作；
 1. 初始阶段，系统随机抽取技能
 2. 掷骰子，人物前进
 3. 前进后，判断是否需要给过路费
 - 现金不足，则破产，结束游戏，人物底色变灰
 4. 使用/不使用技能
 5. 如果金钱充足，可以买地，建房子，升级房子
 6. 若房产属于自己，可以卖房子
 7. 结束回合(只有投掷过骰子，方可结束)

2. 建筑系统

建筑分为以下级别：

- 平地
- 初级房子
- 中级房子
- 高级房子

买地根据不同地皮花钱

¥ 2000 升级一次

3. 金钱系统

起始：15000

支出

- 买地
- 建房子
- 过路费

过路费 = 地皮价格 * 房子增值系数 * 0.1

房子增值系数 = 2^n ，n = 房子升级次数

收入

- 过路费
- 银行卖地

(地皮价格 + 升级花费) * 0.7

4. 技能系统

- 1. 哄抬物价(提高所处位置的地价)
- 2. 贸易战争(降低所处位置的低价)
- 3. 拆迁办(半价拆除)
- 4. 欺诈者(买地8折)

5. 结束清算

- 1. 人物
 - 现金
 - 不动产(地皮价格 + 升级价格)
 - 总资产(现金 + 不动产)
 - 排名

游戏运行

- 1. 将 class 和 recourse 文件夹覆盖一个 cocos 项目后，可编译运行
- 2. 使用当前目录下的 Billionaire.exe

游戏架构

游戏代码抽象分离出三个模块：场景(UI)模块，动画模块以及控制模块。

- 场景(UI)模块
 - 负责场景信息更新，包括当前玩家标识，玩家信息和事件信息显示
- 动画模块
 - 负责动画效果的执行，包括人物移动，技能动画
- 控制模块
 - 负责游戏状态的检测与更新，包括人物状态，地皮状态以及回合状态
 - 人物状态:金钱 / 位置 / 房产
 - 地皮状态:价格 / 升级次数 / 拥有者
 - 回合状态:当前玩家 / 骰子点数(0标志未投) / 技能id / 技能释放标志

游戏逻辑伪代码设计

- 场景模块(Scene)

```
void GameStart(); // 调用动画模块 / 控制系统初始化方法
void RoundStart(){
    /* 显示当前玩家
    * getUserID()
    * 从游戏状态获取当前玩家id
    * 改底色
    */

    /*
    * updatePlayerState();
    */

    /* 技能 bling
    * newSkillID()
    * 从游戏状态获取技能id
    * skillBling();
    * 通知动画模块显示技能 bling 动画
    */
}

void playerLose(); // 玩家破产，底色变灰色

void updatePlayerState() {
    // 更新右上角玩家信息栏
    // 更新金钱 getMoney()
}

void log(stirng msg) {
    // 显示 msg 信息
}

void roundEnd() {
    if (扔了筛子 isRolled()) {
        // 当前玩家回合结束 nextPlayer()
    }
}
```

• 动画模块

```
void initial() {
    //放置人物
}

void skillBling();

void playerMove(int id, int pos, int steps) {
    // 移动
    // 回调 payCharge()
}
```

• 控制系统

- 游戏状态
 - 人物状态
 - 金钱 / 位置 / 房产
 - 地皮状态
 - 价格 / 升级次数 / 拥有者
 - 回合状态
 - 当前玩家 / 骰子点数(0标志未投) / 技能id / 技能释放标志



```
void initial(); // 初始化游戏状态

int getUserID();
int newSkillID(); // random 一下
int getSkillID();
int getMoney(); // 更新当前玩家金钱

int roll() {
    // random 修改当前状态下的骰子点数
    // log(骰子点数)
    // playerMove(int id, int pos, int steps)
    // 更新人物位置
}

void payCharge(){
    // 给过路费
    // log()
    // 更新金钱
    // updatePlayerState()
    // 判断破产(yes的话, 调用UI模块的playerLose(), log())
}

void useSkill() {
    // 更新技能释放标志
    // log()
}

void buyLand() {
    /* if (地无主 && 钱够) {
        更新当前地皮的所有者
        地皮id插入当前人物的房产列表
        扣钱
        log()
        updatePlayerState()
    }
    */
}

void upgradeLand() {
    /*
    if (地属于自己 && 钱够 && 地还能升级) {
        更新地皮升级次数
        扣钱
        log()
        updatePlayerState()
    }
    */
}

void sellLand() {
    /*
    if (地属于自己) {
        更新当前地皮的所有者
        地皮id从当前人物的房产列表中删除
        加钱
        log()
        updatePlayerState()
    }
    */
}

bool isRolled();

void nextPlayer() {
    // 重置为回合初始状态
    // roundStart()
}
```



项目分工

- 徐伟元(场景模块)
- 许倚安(动画模块 + 美工小姐姐)
- 熊永琦(控制模块)

开发细节

由于设计时进行了模块分离，所以在模块实现过程中，每个模块的函数有大部分为调用其他模块函数。因此，在实现过程中，代码书写十分顺畅，在 merge 过程中会有较多问题。下面谈谈各自模块的实现过程中以及 merge 工作中各模块出现的问题：

场景模块(Scene)

1. Layer 底色

由于使用玩家背景底色来标识当前回合玩家，所以需要有一个 layer 来变化底色。考虑到 **层次化** 结构，采用右侧边栏一个 layer，里面包含4个玩家 layer 和一个 log 信息窗口，其中每个玩家 layer 包含玩家头像与金钱数量。

针对底色问题，采用继承 Layer 的 LayerColor 类。

```
// 初始化右侧栏
rightBar = CCLayerColor::create(Color4B::GRAY);
// rightBar info settings
this->addChild(rightBar, 1);
// 初始化玩家头像和金钱模型
for (int i = 0; i < 4; ++i) {
    players[i] = CCLayerColor::create(Color4B(200, 200, 0, 255));
    // player layer info settings
    rightBar->addChild(players[i], 1);

    // 头像
    playerImage[i] = Sprite::create("closeNormal.png");
    // 金钱
    moneyLabel[i] = Label::create("10000", "arial", 15);

    // player and money info settings

    players[i]->addChild(playerImage[i], 1);
    players[i]->addChild(moneyLabel[i], 1);
}

// Log 窗口
logLabel = Label::create("Test", "arial", 15);
// log info settings
rightBar->addChild(logLabel, 1);
```

2. 场景单例(难点)

由于模块分离带来的编程困难

一开始设想将场景类(Scene)狗造成单例模式，这样方便在不同模块间进行调用。但在 merge 过程中，控制模块调用场景模块的 log 函数时，报错场景类内 Label 指针为空指针。经过查阅官方文档，场景类会自动 release，那么如何实现模块调用呢？于是改进为 **实时查询类实例** 来保证当前模块为正在运行模块。

依旧使用 getInstance 方法，但是在每次调用的时候指定调用这个函数而不是在类内部存储一个变量来保存。同时更改场景类的创建代码，默认的 createScene 返回 Scene 类，这里使用 c++ 子类指针可以强制转换为父类指针的特性(丢失部分不用担心，原本代码就是运用原理)，我们在 createScene 时，更改 instance 指针的值，保证场景运行时，其指针指向当前运行场景。

```
static GameMainScene* createScene();

virtual bool init();

static GameMainScene* getInstance() {
    if (instance == nullptr) {
        instance = new GameMainScene();
        instance->init();
    }
    return instance;
}

GameMainScene* GameMainScene::instance = nullptr;

GameMainScene* GameMainScene::createScene() {
    instance = GameMainScene::create();
    return instance;
}
```

动画模块

1. 添加 Sprite

本来预想是动画模块中直接调用场景单例来添加Sprite，但是由于实时查询的问题，导致如果在 initial 中直接 addChild 会重复调用场景类的 init()，只能将 Sprite 创建并存储为 public 变量，然后在场景类中调用添加。

2. 添加技能动画

此处遇到了同1一样的问题，且需要播放动画，于是根据技能的ID将动画提前加好，场景类中直接 addChild。


3. 人物移动

使用了 rpgmaker 的人物创建工具来创建了行走动画，initial时切割好存好四个方向的动画，根据目的地来调用不同的动画实现转向、行走，到达目的地时停止动画并使用 Sequence 来调用 payCharge。

```
auto pay = CallFunc::create([this, id]() {
    BSystemController::getInstance()->payCharge();
    players[id]->getActionManager()->removeAllActionsByTag(1, players[id]);
});

if (pos < 9) {
    auto animation = Animation::createWithSpriteFrames(playerMoving[id * 4 + 3], 0.3f);
    auto animate = RepeatForever::create(Animate::create(animation));
    animate->setTag(1);
    players[id]->runAction(animate);

    int stepped = pos + steps > 9 ? 9 - pos : steps;
    players[id]->runAction(Sequence::create(
        MoveBy::create(steps, Vec2(0, stepped * 64)), pay, nullptr));
    steps -= stepped;
    pos += stepped;
}
```



控制系统模块

控制系统负责统筹管理游戏中的所有数据，以及数据之间的逻辑交互。它通过给 UI 模块提供各种接口，来实现游戏中的各种功能。其定义代码如下：

```

struct PlayerStatus {
    int wealth;
    int location;
    bool isAlive;
    std::vector<int> lands;
    // id is stored as 'index'
    void initial() {
        wealth = INIT_MONEY;
        location = 0;
        isAlive = true;
        lands.clear();
    }
};

struct Land {
    int price;
    int level;
    int owner;

    void initial() {
        level = 0; // no level
        owner = -1; // no owner
    }

    int getCharge() {
        return price * 0.1 * pow(2, level);
    }
};

class BSystemController {
private:

    static BSystemController* _instance;

    PlayerStatus* players = nullptr;
    Land* lands = nullptr;

    int presentPlayer;
    int rollNumer;
    int skillID;
    bool skillUsed;

    //GameMainScene* sceneInstance;
    // AnimationController* aniInstance;

    BSystemController();
    ~BSystemController();

public:

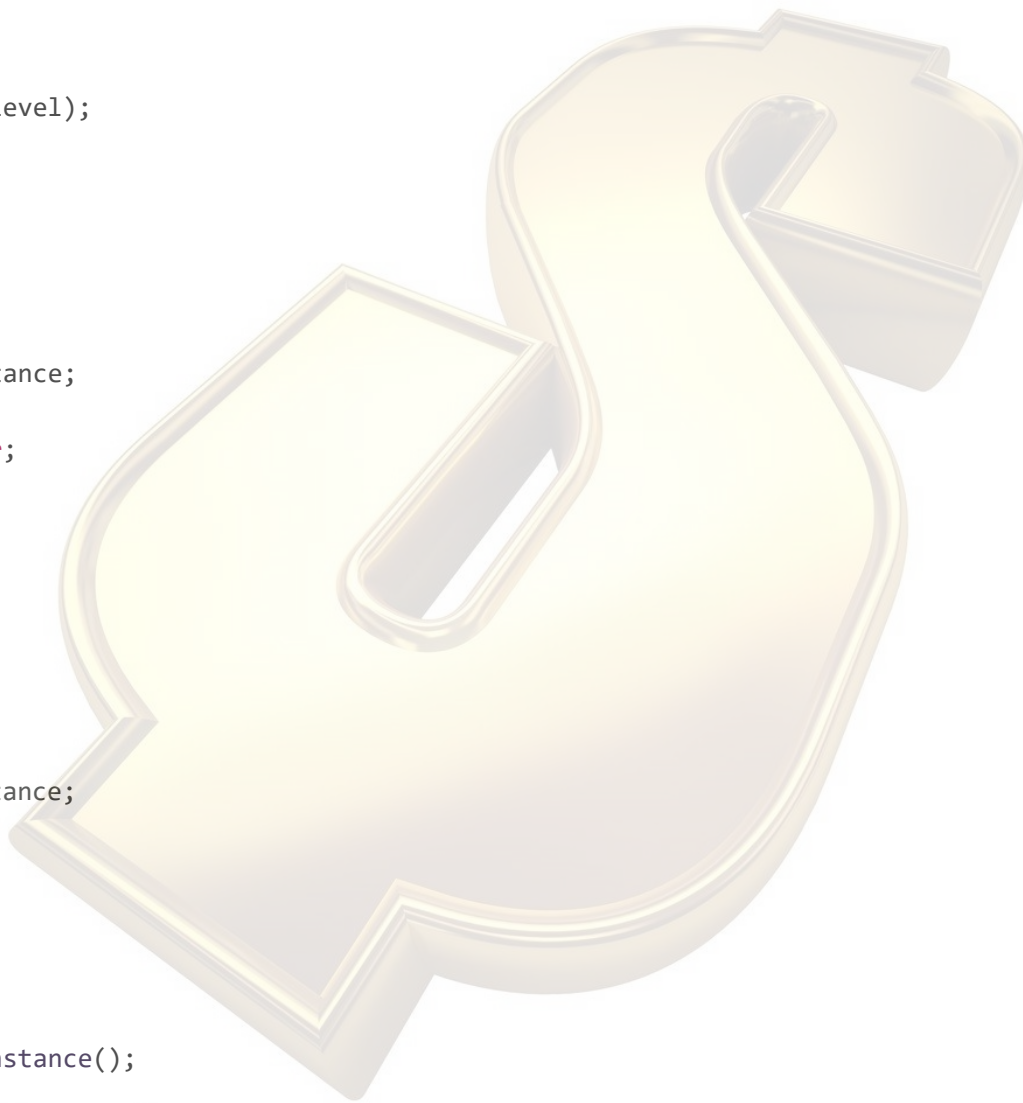
    static BSystemController* getInstance();

    void initial();
    int getUserID();
    int newSkillID();
    int getMoney(int playerID);
    int* getLandsPrice();

    int roll();
    void payCharge();
    void useSkill();
    void buyLand();
    void upgradeLand();
    void sellLand();
    bool isRolled();
    void nextPlayer();

};

```



如代码中所示，在控制系统模块中，我们通过定义 `PlayerStatus` 和 `Land` 结构体来保存单个玩家的状态和每个地皮的状态，并通过两个数组来分别实现对于所有用户和地皮的管理和访问。

遇到的问题

- 无法解析的外部符号

从设计模式上来说，控制系统模块使用了单例模式的设计方法，从而方便维护数据的唯一性。但是在为类设置一个静态私有指针的时候我们发现，在编译的时候，这个指针始终被认为是“无法解析的外部符号”。通过仔细的分析和论证，我们认为这是由于其虽然在编译的分析阶段被写入了符号表，但是没有指向具体的数据实体导致的。类中的普通变量因为有构造函数进行默认的初始化而不会遇到该问题，静态变量由于不会由构造函数进行初始化，从而仅存在了声明（符号表中存在相应的符号），但是缺失了定义（没有办法对该符号进行解析）。所以需要我们自行在cpp文件中添加其的定义如下：

```
BSystemController* BSystemController::_instance = nullptr;
```

- 字符串和数字的加法
写过 C# 或者是 JavaScript 一类语言的同学一定了解，在想要将数字拼接到字符串的时候，最方便的做法就是直接把两者加起来了.....但是基于 C++ 的 cocos 并非如此。在 cocos 中，对一个字符串直接加上数字的行为，cocos 会判定为指针加法。换言之，其把字符串的首地址和数字进行相加，返回的是一个偏移了数位的字符串地址！这给我们项目开发带来了一定的烦恼，最终，我们使用了 c++11 中的字符串转换方法才解决了这个问题。

后记/感想

1. 徐伟元
期末考试完成，立刻投入到这个游戏作业的开发之中，真是紧张刺激。相比较于课上学习的自由度较高的(RPG)游戏，这次我们实现的是一个回合制游戏，一开始还觉得会有些难办，毕竟逻辑组织需要更加严密，尤其是回合值游戏更加依赖状态机。不过，不得不说，小组一起讨论，脑洞大开，看似紧凑一体的逻辑，被剥离出了不同模块。这样就让整个编程变得有意思了。最难过的大概就是 merge 工作了，你的 bug，不，他的，不，我的 orz。
整个课程到这也到了末尾，总之，很感谢老师和ta的付出，很感谢队友了。
2. 许倚安
考完试再来疯狂赶车做基本都忘了C++的语法了，疯狂报类型错误，相对于之前用得比较多的C#，C++基本类型加上cocos的各种内置类型、宏类型，感觉被掏空。对于动画来说，复杂的回调逻辑也是少不了的，跟之前的作业一样，回调的时候一定要注意回调时候拿到的变量还能不能用。当然C++果然还是入门必备吧，相对于C#、java等类型不那么敏感的语言来说，写代码的时候脑海里一定要清楚变量的类型，这种思想是很重要的贯穿了学习的过程，写这个作业的时候有种捡回这种思想的感觉。而且写代码之前还去画地图、做素材、找素材了，突然回到代码的世界有点不适应呢~
3. 熊永琦
本次期末项目带我们回顾了下半学期所学的cocos的内容，以至于大一所学的 C++ 的基础知识。虽然项目和期末复习略有撞车，但是完成过程还是充满成就感的。不过在完成项目的过程中我们能够明显感觉到，相对于使用 C# 的 unity 或是 UWP，C++ 下的 cocos 有着一些难以忽视的问题。首先是其类型转换的不智能，如我之前提到的那样，C++ 对于字符串和其他类型的加法并不会进行转换后拼接的操作，而是会将其视为对于指针的操作，这让原本可以简单解决的代码变得复杂，削减了代码的可读性。同时，C++ 的宏操作也令人苦恼，较为落后的回调设计，以及漫长的编译部署时间也让人感到头大。但是 C++ 并非全是坏处，其最大的优点显而易见——更优秀的运行速度。在 Unity 和 UWP 中，我们是能够经常性遇见优化不足导致的性能瓶颈的，卡顿和延迟可以说是家常便饭。但是在 C++ 书写的 cocos 中，卡顿似乎成为了最不需要担心的事——一切都是非常的流畅、顺滑，这样的高性能表现对于手机这种硬件捉襟见肘的平台来说是十分重要的。可是在对编程者的友好程度上，C++ 还是需要再学习一个。