新建Cocos项目：

cocos new –l cpp –d ① –p ② ③

cocos 是cocos2dx提供的一个命令集，包括新建项目，编译，发布，运行等命令

new 是其中一个子命令，用于新建一个cocos项目

-l language，选择后面开发cocos的语言，这里我们使用cpp，即C++

以下为可选项：

-d directory，指定项目存放地，①为你自己选择的目录，默认项目创建在当前目录中

-p package，指定java包名，②是你自己的包名称，默认是cocos官方包名

③ 项目名称，默认是MyCppGame，推荐每个单词首字母大写

【对于舍不得重装系统的童鞋：

1.安装python，在清单中最后一项默认是红叉，点击选择第一项让它自动设置环境变量

2.设置cocos环境变量(python必须已经安装并配置成功)：

a.进入cocos2d-x3.0目录，输入【setup.py】，它是一个python脚本，用于设置cocos所需命令的环境变量，当前可以一路回车，完成后重启终端即可。

新项目下的目录：

Classes：C++源文件

Resources：资源文件

Hello项目中有四个部分：

Menu：字体菜单

Sprite：精灵

Menu: 精灵菜单

FPS： 渲染顶点数

渲染批次

帧率

项目中的术语（渲染树）：

Director：导演。资源管理

Scene： 场景。

Layer： 层

Sprite：精灵

添加背景图片：

* 1. 找到AppDelegate，在找到摄像师之后，添加资源路径：

//告诉道具师，道具的位置

FileUtils::getInstance()->addSearchPath("ui");

* 1. 回到HelloWordScene，进入init添加加载精灵集文件到内存代码

//加载精灵集到内存

SpriteFrameCache::getInstance()->addSpriteFramesWithFile("shoot\_background.plist");

//创建背景精灵

auto bg = Sprite::createWithSpriteFrameName("background.png");

//添加到当前层中，【->】表示左侧是一个指针

this->addChild(bg);

//修正位置

bg->setPosition(400, 250);

自定义一个层类

1. 创建基本的C++类需要注意的

C++的类推荐用头文件来声明类成员，用源文件来实现类成员

头文件声明：

1. 因为我们的类需要使用Cocos的一些已定义类和API，所以在文件开始，导入相关头文件和引入名称空间：

#include "cocos2d.h"

//using namespace cocos2d;

USING\_NS\_CC;

1. C++中类定义的语法：

class GameScene : public Layer {

public:

//static GameScene create();

//通过调用create函数，返回GameScene的对象

CREATE\_FUNC(GameScene);

//create调用init来初始化层中的内容

bool init();

//返回一个包含GameScene这个Layer的场景

static Scene \*createScene();

};

注意继承的语法与公有继承的概念

注意public的位置及其冒号的使用

注意类内部定义函数的语法

注意返回指针的函数的语法

注意类声明结束之后的分号

1. 源文件的语法

Scene \*GameScene::createScene() {

//创建我们的层

auto layer = GameScene::create();

//创建一个普通的场景

auto scene = Scene::create();

//将我们的层加到这个场景中

scene->addChild(layer);

return scene;

}

注意函数中类外实现需要添加类名限定

注意auto的使用，必须在声明同时初始化

【注意：一定是在Classes上右击-添加-新建项，存放目录指定到项目下面的Calsses中

1. 坐标与锚点的概念
   1. 坐标控制元素的锚点位置，规定左下角为原点，提供x和y两个方向
   2. 而锚点是相对于元素宽高的一个比值，规定左下角为(0, 0)，右上角为(1, 1)，默认为(0.5, 0.5)即中心
2. 设置运行时窗口大小

//设置窗口大小

glview->setFrameSize(480, 640);

1. 背景的移动
   1. 添加两张背景

auto bg2 = Sprite::createWithSpriteFrameName("background.png");

this->addChild(bg2, -1, 2);

bg2->setAnchorPoint(Point::ZERO);

//getContentSize()得到bg1的内容大小

bg2->setPositionY(bg1->getContentSize().height);

//纹理采用抗锯齿

bg2->getTexture()->setAliasTexParameters();

* 1. 通过scheduleUpdate函数每帧都周而复始的调用update函数

//保证每一帧都调用一次update方法

this->scheduleUpdate();

* 1. 每帧中都更新背景图的位置

void GameScene::update(float dt) {

//背景动起来

//父容器通过添加子元素时绑定的tag来获取这个子元素

auto bg1 = this->getChildByTag(1);

auto bg2 = this->getChildByTag(2);

bg1->setPositionY(bg1->getPositionY() - 2);

//第二张背景图始终保持在第一张图的上面

bg2->setPositionY(bg1->getPositionY() + bg1->getContentSize().height);

//当第二张图移动到底部时，将第一张图重新放上来

if(bg2->getPositionY() <= 0) {

bg1->setPositionY(0);

}

}

1. 添加飞机
   1. 获取屏幕大小

//获取屏幕的宽高

auto visibleSize = Director::getInstance()->getVisibleSize();

* 1. 设置飞机的位置到屏幕底部的中央

float x = visibleSize.width/2;

float y = hero->getContentSize().height/2;

hero->setPosition(x,y);

* 1. 添加动画

//添加飞机飞行动画

auto animation = Animation::create();

//有哪些图片参与动画

animation->addSpriteFrame(SpriteFrameCache::getInstance()->getSpriteFrameByName("hero1.png"));

animation->addSpriteFrame(SpriteFrameCache::getInstance()->getSpriteFrameByName("hero2.png"));

//切换的频率

animation->setDelayPerUnit(0.2f);

//切换的周期数

animation->setLoops(-1);

* 1. 将动画封装为动作让飞机调用

//将动画转换为动作

auto action = Animate::create(animation);

//执行动作

hero->runAction(action);

1. 拖动飞机移动
   1. Cocos的触摸事件处理

//添加触摸事件

//创建单点触摸事件监听器

auto listener = EventListenerTouchOneByOne::create();

//为飞机添加事件

this->getEventDispatcher()->addEventListenerWithSceneGraphPriority(listener, hero);

* 1. 对触摸事件的分解回调

listener->onTouchBegan = [hero](Touch \*touch, Event \*){

log("==============began");

return true;

};

listener->onTouchMoved = [hero](Touch \*touch, Event \*){

log("==============move");

};

listener->onTouchEnded = [](Touch \*, Event \*){

// log("==============end");

};

* 1. 判定触摸点是否在飞机的绘制矩形之内

//如何知道，点击发生在飞机上

//通过touch对象获得触摸点的坐标

auto touchPos = touch->getLocation();

bool isContain = hero->getBoundingBox().containsPoint(touchPos);

return isContain;

* 1. 为类添加一个成员变量vec用于跨函数传递起始触摸点与飞机坐标的向量差，注意在构造函数中为变量初始化，不能在类声明中直接初始化。
  2. 记录触摸点坐标与飞机锚点坐标之间的向量差

//获取触摸点与飞机锚点的向量差

this->vec = hero->getPosition() - touchPos;

* 1. 在移动手指时，同时更新飞机锚点坐标（触摸坐标+向量差）

//飞机要跟随手指移动

hero->setPosition(touch->getLocation()+this->vec);

* 1. 手指抬起时，清零vec

1. 适屏处理
   1. Cocos中有三种尺寸
      1. 资源尺寸
      2. 设计尺寸
      3. 物理尺寸
   2. Cocos提供的五种适屏策略
      1. Fit拉伸。优点：无黑边，无裁剪；缺点：变形
      2. Noborder无黑边。优点：无黑边；缺点：裁剪不可控
      3. Showall全显示。优点：无裁剪；缺点：黑边严重
      4. FixedHeight高度无裁剪。优点：高度刚好与物理高度一致；缺点：宽度有可能有黑边或者裁剪。
      5. fixedWidth宽度无裁剪。

//适屏，参数依次是：设计尺寸的宽，高，设计尺寸中各种物理尺寸转化过程中的适屏策略

glview->setDesignResolutionSize(480, 640, ResolutionPolicy::FIXED\_WIDTH);

1. 飞机的边界控制

在移动飞机之前，对坐标的x和y分别进行修正

//飞机要跟随手指移动

Point nextPos = touch->getLocation()+this->m\_vec;

//边界拦截

float minX = hero->getContentSize().width/2;

float maxX = visibleSize.width-minX;

float minY = hero->getContentSize().height/2;

float maxY = visibleSize.height-minY;

float x = MAX(minX, MIN(nextPos.x, maxX));

float y = MAX(minY, MIN(nextPos.y, maxY));

hero->setPosition(x, y);

1. 子弹的创建
   1. 单独创建Bullet类，以便对sprite进行扩展
   2. 创建子弹时需要知道飞机的坐标来确定初始位置，所以默认的create不能满足要求需要自己实现

Bullet \*Bullet::create(Point pos) {

//创建一个新对象

auto bullet = new Bullet();

//如果对象创建成功，并且初始化成功

if (bullet && bullet->init(pos)) {

//放入自动释放池

bullet->autorelease();

return bullet;

}

//删除分配空间

delete bullet;

bullet = NULL;

return NULL;

}

bool Bullet::init(Point pos) {

//调用的是init而不是create

Sprite::initWithSpriteFrameName("bullet1.png");

this->setPosition(pos);

return true;

}

* 1. 为公用数据提供一个统一的头文件，在需要访问这些数据时包含这个头文件

//获取可见窗口大小

#define VISIBLE\_SIZE Director::getInstance()->getVisibleSize()

//子弹每次移动的距离

#define BULLET\_STEP 6

* 1. 在GameScene中使用一个迭代器，定期生成子弹对象，方便后期访问，在创建的同时放入到一个Vector容器中

//自由的定时器，被调函数名称自定义，间隔时间自定义

//注意，函数名要selector包裹，函数签名必须是void返回和float单参数

schedule(schedule\_selector(GameScene::createBullet), 0.1f);

void GameScene::createBullet(float) {

//通过添加到层中的tag找到飞机对象

auto hero = this->getChildByTag(10);

//根据飞机坐标创建子弹

auto bullet = Bullet::create(hero->getPosition() + Point(0, hero->getContentSize().height/2));

//添加到层中，子弹将被渲染（绘制）出来

this->addChild(bullet);

//同时，添加到集合中，以便后续访问与操作

this->m\_bullets.pushBack(bullet);

}

【小知识：Vector是C++11中新增的集合，它不用我们参与内存释放和初始化问题，并且它支持泛型（一种将运行时检查提前到编译时检查的有效机制）。另外C++11还提供了一种简化的遍历集合的循环for(成员类型 变量名 : 集合名词)】

* 1. 中update中移动子弹集合中的每一个子弹的坐标，达到子弹运动的效果

//专门遍历集合的特殊for循环，类似C#中foreach

for(auto bullet : this->m\_bullets) {

//更新子弹坐标，让它向上移动

bullet->setPositionY(bullet->getPositionY() + BULLET\_STEP);

}

* 1. 同时，可以判断子弹是否飞出屏幕，从性能角度考虑，应该移除飞出屏幕的子弹。但是在遍历集合过程中，是不能够修改集合状态的，所以，利用一个临时集合来保存要移除的子弹，待全部判断完之后，统一从集合中移除。

//让子弹飞起来

//用来存放飞出屏幕，需要被移除的子弹

Vector<Bullet \*> removableBullets;

//专门遍历集合的特殊for循环，类似C#中foreach

for(auto bullet : this->m\_bullets) {

//更新子弹坐标，让它向上移动

bullet->setPositionY(bullet->getPositionY() + BULLET\_STEP);

//获取子弹移动的上限

float maxY = VISIBLE\_SIZE.height+bullet->getContentSize().height/2;

if (bullet->getPositionY() >= maxY)

{

//记录飞出上限的子弹

removableBullets.pushBack(bullet);

//从父容器中移除这颗子弹（将不再被渲染（不可见））

this->removeChild(bullet);

}

}

for(auto bullet : removableBullets) {

//从子弹集合中移除被记录需要移除的子弹，避免对这些子弹浪费运算

this->m\_bullets.eraseObject(bullet);

}

//清空临时集合

removableBullets.clear();

1. 创建小敌机
   1. 同创建子弹类似，通过定时器方法循环创建敌机并添加到存放敌机的集合中。在update中遍历每一个敌机，并移动它们，如果有敌机出界则标记起来。待遍历完成之后，统一销毁

//===============创建敌机（init函数中）

schedule(schedule\_selector(GameScene::createSmallEnemy), 0.8f);

//创建小敌机的函数

void GameScene::createSmallEnemy(float) {

auto enemy = Sprite::createWithSpriteFrameName("enemy1.png");

//随机小数\*（max-min+1）+min+1

float minX = enemy->getContentSize().width/2;

float maxX = VISIBLE\_SIZE.width-minX;

float x = rand() % (int)(maxX-minX)+minX;

float y = VISIBLE\_SIZE.height+enemy->getContentSize().height/2;

enemy->setPosition(x, y);

this->addChild(enemy, 4);

this->m\_enemies.pushBack(enemy);

}

//让敌机掉下来(在update函数中)

Vector<Enemy \*> removableEnemies;

for(auto enemy : this->m\_enemies) {

enemy->setPositionY(enemy->getPositionY()-SMALL\_ENEMY\_STEP);

if (enemy->getPositionY() <= -enemy->getContentSize().height/2)

{

removableEnemies.pushBack(enemy);

this->removeChild(enemy);

}

}

for (auto enemy : removableEnemies)

{

this->m\_enemies.eraseObject(enemy);

}

removableEnemies.clear();

1. 提取敌机类

敌机的后续功能不是单纯的Sprite可以满足，我们必须在此基础上进行扩展，以满足需求，先新建Enemy类，继承自Sprite，保证重构后上述功能能够正常执行。

* 1. 在init中事先准备好爆炸动画，以便后期敌机被击中时调用

auto animation = Animation::create();

for (int i = 1; i <= 4; i++)

{

//拼接字符串，参见C中的printf函数

auto png = StringUtils::format("enemy1\_down%d.png", i);

//根据精灵帧名称从精灵帧缓存中获取精灵帧

auto spriteFrame = SpriteFrameCache::getInstance()->getSpriteFrameByName(png);

animation->addSpriteFrame(spriteFrame);

}

//默认动画只执行一次，所以这里缺省

animation->setDelayPerUnit(0.2f);

//因为不会立即执行，所以先在动画缓存中缓存起来

AnimationCache::getInstance()->addAnimation(animation, "SmallEnemyDown");

* 1. 实现被击中时调用的down函数。为了保证动画执行完之后及时删除敌机，这里引入CallFuncN类和Sequence类

//敌机被击中时调用的函数，主要功能是执行爆炸动画和从父元素中移除自己（不再绘制）

void Enemy::down() {

//根据名称从动画缓存中找回动画

auto animation = AnimationCache::getInstance()->getAnimation("SmallEnemyDown");

auto animate = Animate::create(animation);

//CallFuncN运行我们在动画序列中插入代码，其中N代表执行这个动作的元素自身

auto callFunc = CallFuncN::create([](Node \*node){

//将自己从父容器中移除

node->removeFromParentAndCleanup(true);

});

//Sequen保证了动画执行的先后顺序，末尾的NULL表示动画序列的终止，不可省略

this->runAction(Sequence::create(animate, callFunc, NULL));

}

1. 敌机类型
   1. 新增一个变量来表示敌机类型，并且中init中根据类型不同加载不同的精灵帧

this->m\_type = type;

switch (type)

{

case SMALL\_ENEMY:

//调用的事init而不是create

Sprite::initWithSpriteFrameName("enemy1.png");

break;

case MIDDLE\_ENEMY:

//调用的事init而不是create

Sprite::initWithSpriteFrameName("enemy2.png");

break;

case BIG\_ENEMY:

//调用的事init而不是create

Sprite::initWithSpriteFrameName("enemy3\_n1.png");

break;

default:

break;

}

* 1. 并且缓存另外两种敌机的爆炸动画
  2. 而大敌机还有飞行动画

//大飞机的飞行动画

auto flyAnim = Animation::create();

flyAnim->addSpriteFrame(SpriteFrameCache::getInstance()->getSpriteFrameByName("enemy3\_n1.png")); flyAnim->addSpriteFrame(SpriteFrameCache::getInstance()->getSpriteFrameByName("enemy3\_n2.png"));

flyAnim->setDelayPerUnit(0.2f);

flyAnim->setLoops(-1);

this->runAction(Animate::create(flyAnim));

* 1. 在GameScene中添加三个定时器分别负责循环创建三种类型的敌机

schedule(schedule\_selector(GameScene::createSmallEnemy), 0.8f, -1, 3);

schedule(schedule\_selector(GameScene::createMiddleEnemy), 2, -1, 6);

schedule(schedule\_selector(GameScene::createBigEnemy), 20, -1, 12);

//创建敌机的函数

void GameScene::createEnemyByType(EnemyType type) {

auto enemy =Enemy::create(type);

//随机小数\*（max-min+1）+min+1

float minX = enemy->getContentSize().width/2;

float maxX = VISIBLE\_SIZE.width-minX;

float x = rand() % (int)(maxX-minX)+minX;

float y = VISIBLE\_SIZE.height+enemy->getContentSize().height/2;

enemy->setPosition(x, y);

this->addChild(enemy, 4);

this->m\_enemies.pushBack(enemy);

}

void GameScene::createSmallEnemy(float) {

createEnemyByType(SMALL\_ENEMY);

}

void GameScene::createMiddleEnemy(float) {

createEnemyByType(MIDDLE\_ENEMY);

}

void GameScene::createBigEnemy(float) {

createEnemyByType(BIG\_ENEMY);

}

* 1. 在Enemy中为不同的敌机设置不同的单步移动距离和HP

switch (type)

{

case SMALL\_ENEMY:

//调用的事init而不是create

Sprite::initWithSpriteFrameName("enemy1.png");

this->m\_enemyStep = SMALL\_ENEMY\_STEP;

this->m\_HP = 1;

break;

case MIDDLE\_ENEMY:

//调用的事init而不是create

Sprite::initWithSpriteFrameName("enemy2.png");

this->m\_enemyStep = MIDDLE\_ENEMY\_STEP;

this->m\_HP = 4;

break;

case BIG\_ENEMY:

{

//调用的事init而不是create

Sprite::initWithSpriteFrameName("enemy3\_n1.png");

this->m\_enemyStep = BIG\_ENEMY\_STEP;

this->m\_HP = 8;

}

break;

default:

break;

}

* 1. 在GameScene的update中，修改敌机的移动，不再是固定的值而是敌机自己保存的值

//============让敌机掉下来

Vector<Enemy \*> removableEnemies;

for(auto enemy : this->m\_enemies) {

enemy->setPositionY(enemy->getPositionY()-enemy->getEnemyStep());

if (enemy->getPositionY() <= -enemy->getContentSize().height/2)

{

removableEnemies.pushBack(enemy);

this->removeChild(enemy);

}

}

* 1. 同样在GameScene的update中，碰撞检测时，敌机不再是直接销毁，而是减掉一滴血，如果HP为零，才执行爆炸销毁

//判断相撞：两个矩形区域是否有交集

if(enemy->getBoundingBox().intersectsRect(bullet->getBoundingBox())) {

removableBullets.pushBack(bullet);

this->removeChild(bullet);

enemy->setHP(enemy->getHP() - 1);

if (enemy->getHP() <= 0)

{

removableEnemies.pushBack(enemy);

enemy->down();

} else {

enemy->hit();

}

}

1. 敌机受到子弹攻击的代码处理
   1. 同敌机爆炸动画一样，敌机受伤动画也需要中init中先缓存起来

//=========受伤的动画

animation = Animation::create();

//根据精灵帧名称从精灵帧缓存中获取精灵帧

animation->addSpriteFrame(SpriteFrameCache::getInstance()->getSpriteFrameByName("enemy2\_hit.png"));

animation->addSpriteFrame(SpriteFrameCache::getInstance()->getSpriteFrameByName("enemy2.png"));

//默认动画只执行一次，所以这里缺省

animation->setDelayPerUnit(0.2f);

//因为不会立即执行，所以先在动画缓存中缓存起来

AnimationCache::getInstance()->addAnimation(animation, "MiddleEnemyHit");

* 1. 敌机受到子弹攻击一定会减少血量，然后根据血量是否耗尽去分别执行爆炸或者受伤动画

//敌机被子弹击中之后，敌机的反应

void Enemy::hit() {

//血量一定下降

this->m\_HP--;

//如果没有血量了，则调用爆炸函数，并退出此函数

if (this->m\_HP <= 0)

{

this->down();

return;

}

//获取受伤动画并执行

Animation \*animation;

switch (this->m\_type)

{

case MIDDLE\_ENEMY:

animation = AnimationCache::getInstance()->getAnimation("MiddleEnemyHit");

break;

case BIG\_ENEMY:

animation = AnimationCache::getInstance()->getAnimation("BigEnemyHit");

break;

default:

break;

}

this->runAction(Animate::create(animation));

}

1. 用定时器创建道具
   1. 参见敌机的创建以及敌机与子弹的碰撞检测，这里也需要中GameScene中新增一个记录所有创建出来的道具的集合m\_ufos

//创建道具

void GameScene::createUfo1(float) {

auto ufo = Sprite::createWithSpriteFrameName("ufo1.png");

//设置随机水平位置坐标

float minX = ufo->getContentSize().width/2;

float maxX = VISIBLE\_SIZE.width-minX;

float x = rand() % (int)(maxX-minX)+minX;

float y = VISIBLE\_SIZE.height+ufo->getContentSize().height/2;

ufo->setPosition(x, y);

this->addChild(ufo, 4);

this->m\_ufos.pushBack(ufo);

//移动道具（下-上-下-销毁）

auto move1 = MoveBy::create(0.5f, Point(0, -300));

auto move2 = MoveBy::create(0.5f, Point(0, 300));

auto move3 = MoveTo::create(1.2f, Point(x, -ufo->getContentSize().height/2));

auto callFunc = CallFuncN::create([](Node \*node){

node->removeFromParentAndCleanup(true);

});

ufo->runAction(Sequence::create(move1, move2, move3, callFunc, NULL));

}

* 1. 中update中，遍历道具集合进行与英雄飞机的碰撞检测

//道具与英雄的碰撞

Vector<Sprite \*> removableUfos;

for (auto ufo : this->m\_ufos)

{

if (ufo->getBoundingBox().intersectsRect(hero->getBoundingBox()))

{

//如果是子弹道具就切换子弹

//如果是炸弹道具就更新左下角的显示

this->removeChild(ufo);

removableUfos.pushBack(ufo);

}

}

for(auto ufo : removableUfos) {

this->m\_ufos.eraseObject(ufo);

}

removableUfos.clear();

//道具与英雄的碰撞结束

1. 实现子弹的切换
   1. 提取子弹类，用于根据不同类型选择不同的子弹图片

class Bullet : public Sprite

{

public:

Bullet();

static Bullet \*create(BulletType type);

bool init(BulletType type);

private:

BulletType m\_type;

};

* 1. 在GameScene中添加两个变量，分别控制创建子弹的类型和控制双子弹的发射次数

//是否切换到双子弹

bool m\_isDoubleBullet;

//双子弹的发射次数

int m\_doubleBulletNumber;

* 1. 在update的道具与英雄飞机的碰撞检测中，设置控制变量为true，并将双子弹次数填满

//如果是子弹道具就切换子弹

this->m\_isDoubleBullet = true;

//加满双子弹数量

this->m\_doubleBulletNumber = 30;

* 1. 的createBullet中，根据控制变量的状态来分别处理创建单子弹和双子弹的代码，实现子弹的切换

//创建子弹的函数

void GameScene::createBullet(float) {

//通过添加到层中的tag找到飞机对象

auto hero = this->getChildByTag(10);

if (this->m\_isDoubleBullet)

{

//创建左侧的子弹

auto leftBullet = Bullet::create(DOUBLE\_BULLET);

leftBullet->setPosition(hero->getPosition()+ Point(-hero->getContentSize().width/3, 0));

//添加到层中，子弹将被渲染（绘制）出来

this->addChild(leftBullet, 0);

//同时，添加到集合中，以便后续访问与操作

this->m\_bullets.pushBack(leftBullet);

//创建右侧的子弹

…

} else {

//单子弹的创建

…

}

}

* 1. 并且每创建一次双子弹，减少一次计数。当计数变量为0时，切换状态变量为false

//创建一次双子弹，计数减一

this->m\_doubleBulletNumber--;

//次数用完之后，自动切换回单子弹

if (this->m\_doubleBulletNumber <= 0)

{

this->m\_isDoubleBullet = false;

}

1. 分值的计算与显示
   1. 在敌机类中添加分值变量，记录每种敌机被销毁时玩家应该得到的奖励分值
   2. 在GameScene中添加一个变量，用于存放整个游戏过程中累计的分值
   3. 在init函数中，添加两个Label分别显示提示文字和当前累计分值。在使用【font.fnt】文件之前，需要中AppDelegate中添加【font】到搜索路径

FileUtils::getInstance()->addSearchPath("font");

//==============显示分数

//固定内容选用了系统字体

auto lblScore = Label::createWithSystemFont("Score:", "Arial", 36);

lblScore->setPosition(40, VISIBLE\_SIZE.height-40);

lblScore->setAnchorPoint(Point::ANCHOR\_MIDDLE\_LEFT);

lblScore->setColor(Color3B::BLACK);

this->addChild(lblScore, 99); //设置字体颜色

//分值会变化，选用了它提供的fnt字体文件

auto lblScoreNumber = Label::createWithBMFont("font.fnt", "0");

lblScoreNumber->setPosition(lblScore->getContentSize().width + 45, VISIBLE\_SIZE.height-40);

lblScoreNumber->setAnchorPoint(Point::ANCHOR\_MIDDLE\_LEFT);

lblScoreNumber->setColor(Color3B::BLACK);

this->addChild(lblScoreNumber, 99, 90);

* 1. 在update中当敌机被子弹销毁，将飞机对应的奖励分值累加到变量中，并同步label的显示

if (enemy->getHP() <= 0)

{

//累加分钟并更新label的显示

this->m\_score += enemy->getScore();

auto lbl = (Label \*) this->getChildByTag(90);

lbl->setString(StringUtils::format("%d", this->m\_score));

removableEnemies.pushBack(enemy);

}

1. 炸弹道具的处理
   1. 抽取出道具类，用于记住自己的道具类型（两种值的枚举）
   2. 在GameScene的init中添加一个精灵菜单和label，分别显示炸弹图标和炸弹数量

//=============左下角显示炸弹数

auto spriteBomb = Sprite::createWithSpriteFrameName("bomb.png");

//创建了一个精灵菜单项，参数（平时显示的精灵，被点击时显示的精灵， 被点击时触发的代码）

auto menuItemBomb = MenuItemSprite::create(spriteBomb, spriteBomb, [](Ref \*){

//当触摸炸弹图标时，触发全屏飞机的爆炸并记分

});

menuItemBomb->setPosition(40, 40);

menuItemBomb->setAnchorPoint(Point::ANCHOR\_MIDDLE\_LEFT);

//菜单项不可直接添加到层中渲染，需要菜单容器进行封装

auto menu = Menu::create();

menu->addChild(menuItemBomb);

//默认菜单的坐标中中央，这里不方便定位，所以置零，让菜单项自己定位

menu->setPosition(Point::ZERO);

this->addChild(menu, 99); //注意Zorder

//炸弹数会变化，选用类它提供的fnt字体

auto lblBombCount = Label::createWithBMFont("font.fnt", "X0");

lblBombCount->setPosition(menuItemBomb->getContentSize().width+45, 40);

lblBombCount->setAnchorPoint(Point::ANCHOR\_MIDDLE\_LEFT);

lblBombCount->setColor(Color3B::BLACK);

this->addChild(lblBombCount, 99, 91);

1. 炸弹道具的显示与触发
   1. 在道具与飞机的碰撞中判断到如果是炸弹道具更新炸弹显示数量

switch (ufo->getType())

{

//如果是子弹道具就切换子弹

case UFO\_SUPER\_BULLET:

this->m\_isDoubleBullet = true;

//加满双子弹数量

this->m\_doubleBulletNumber = 30;

break;

//如果是炸弹道具就更新左下角的显示

case UFO\_SUPER\_BOMB:

{

this->m\_bombCount++;

auto lbl = (Label \*)this->getChildByTag(91);

lbl->setString(StringUtils::format("X%d", this->m\_bombCount))

}

break;

default:

break;

}

* 1. 提取出updateBombCount函数，提升炸弹数显示算法：
     1. 当没有炸弹时，左下角炸弹图标（菜单）和标签均不显示
     2. 当只有一颗炸弹时，只有图标显示
     3. 当有多颗炸弹时，图标和标签均显示，并更新标签数量

void GameScene::updateBombLabel() {

//先找到菜单，再通过菜单找到精灵菜单项

auto menu = this->getChildByTag(101);

auto item = menu->getChildByTag(100);

//获得标签，思考为什么要强制类型转换

auto lbl = (Label \*)this->getChildByTag(91);

if (this->m\_bombCount <= 0)

{

item->setVisible(false);

lbl->setVisible(false);

} else if (this->m\_bombCount == 1)

{

item->setVisible(true);

lbl->setVisible(false);

} else {

item->setVisible(true);

lbl->setVisible(true);

//别忘了更新炸弹数

lbl->setString(StringUtils::format("X%d", this->m\_bombCount));

}

}

* 1. 在炸弹菜单的触发回调中，触发炸弹的全屏爆炸

//创建了一个精灵菜单项，参数（平时显示的精灵，被点击时显示的精灵， 被点击时触发的代码）

auto menuItemBomb = MenuItemSprite::create(spriteBomb, spriteBomb, [=](Ref \*){

//当触摸炸弹图标时，触发全屏飞机的爆炸并记分

this->m\_bombCount--;

//别忘了更新左下角显示

this->updateBombLabel();

//销毁敌机集合中所有的敌机，并清空集合

for (auto enemy : this->m\_enemies)

{

enemy->down();

//累加分值并更新显示

this->m\_score += enemy->getScore();

}

this->m\_enemies.clear();

//更新分值显示

lblScoreNumber->setString(StringUtils::format("%d", this->m\_score));

});

1. Android真机发布
   1. 发布环境确保：
      1. 安装JDK，并配置JAVA\_HOME环境变量
      2. 安装Ant
      3. 安装Android-SDK，并通过SDK Manager下载
      4. 安装Android-NDK
      5. 重新执行cocos2d-x 3.0文件夹中的setup.py，并按提示完成配置
   2. 更新Android.mk，添加自己cpp文件
   3. 执行build\_native.py，将C++代码混编到Android平台
   4. cocos run –p android 将项目在android中运行（运行之前完成打包）
2. 开始场景
   1. 新建一个开始场景类，具备基本场景类的函数

class WelcomeScene : public Layer

{

public:

bool init();

CREATE\_FUNC(WelcomeScene);

static Scene \*createScene();

};

* 1. 将游戏中用到的【缓存】都迁移到该类的init中

//用当前时间作为随机数的种子

srand((unsigned int)time(0)); SpriteFrameCache::getInstance()->addSpriteFramesWithFile("shoot\_background.plist"); SpriteFrameCache::getInstance()->addSpriteFramesWithFile("shoot.plist");

//===========爆炸的动画

auto animation = Animation::create();

for (int i = 1; i <= 4; i++) {

//拼接字符串，参见C中的printf函数

auto png = StringUtils::format("enemy1\_down%d.png", i);

//根据精灵帧名称从精灵帧缓存中获取精灵帧

auto spriteFrame = SpriteFrameCache::getInstance()->getSpriteFrameByName(png);

animation->addSpriteFrame(spriteFrame);

}

//默认动画只执行一次，所以这里缺省

animation->setDelayPerUnit(0.2f);

//因为不会立即执行，所以先在动画缓存中缓存起来

AnimationCache::getInstance()->addAnimation(animation, "SmallEnemyDown");

…

* 1. 在小飞机加载动画执行两次后跳转场景

animation = Animation::create();

for (int i = 0; i < 4; i++)

{

auto png = StringUtils::format("game\_loading%d.png", i+1);

animation->addSpriteFrame(SpriteFrameCache::getInstance()->getSpriteFrameByName(png));

}

animation->setDelayPerUnit(0.2f);

animation->setLoops(2);

auto animate = Animate::create(animation);

auto callFunc = CallFunc::create([](){

//跳转场景的代码

auto scene = GameScene::createScene();

Director::getInstance()->replaceScene(scene);

});

loading->runAction(Sequence::create(animate, callFunc, NULL));

1. 游戏结束场景
   1. 创建基本的游戏结束场景
   2. 在GameScene的init中进行敌机与飞机的碰撞检测，如果相撞则执行飞机爆炸动画后跳转到结束场景

if (enemy->getBoundingBox().intersectsRect(hero->getBoundingBox()))

{

enemy->down();

removableEnemies.pushBack(enemy);

auto animation = Animation::create();

for (int i = 1; i <= 4; i++)

{

//拼接字符串，参见C中的printf函数

auto png = StringUtils::format("hero\_blowup\_n%d.png", i);

//根据精灵帧名称从精灵帧缓存中获取精灵帧

auto spriteFrame = SpriteFrameCache::getInstance()->getSpriteFrameByName(png);

animation->addSpriteFrame(spriteFrame);

}

//默认动画只执行一次，所以这里缺省

animation->setDelayPerUnit(0.2f);

auto animate = Animate::create(animation);

auto callFunc = CallFunc::create([=](){

auto scene = GameOverScene::createScene(this->m\_score);

Director::getInstance()->replaceScene(scene);

});

hero->stopAllActions();

hero->runAction(Sequence::create(animate, callFunc, NULL));

}

注意：将GameScene累计的得分也要传递到GameOverScene中，以便显示与处理

1. 记录最高分
   1. Cocos提供了很多方便我们将数据保存到文件的方法，其中UserDefault是比较简单的实现
   2. 从文件中读取历史最高分，如果当前分值比历史最高分高，则更新

//如果读取到的最高分没有当前的得分值高。那么，更新最高得分且写入到配置文件中，为了引起玩家的注意，添加了缩放动画

if (highScore<score)

{

highScore = score;

UserDefault::getInstance()->setIntegerForKey("HIGHSCORE", highScore);

lblFinalScore->setString(StringUtils::format("%d", highScore));

auto scaleIn = ScaleTo::create(0.5f, 1.5);

auto scaleOut = ScaleTo::create(0.5f, 1);

lblFinalScore->runAction(Sequence::create(scaleIn, scaleOut, NULL));

}

1. 重新开始
   1. 只需重新跳转到GameScene即可实现

auto menuItemBack = MenuItemSprite::create(back, back, [](Ref \*){

auto scene = GameScene::createScene();

Director::getInstance()->replaceScene(scene);

});

1. 暂停
   1. 在GameScne的init中为屏幕右上角添加暂停和继续菜单项，注意这里处理点击事件的写法不再是使用lambda表达式，而是CC\_CALLBACK\_1

//===========暂停按钮

auto pauseNor = Sprite::createWithSpriteFrameName("game\_pause\_nor.png");

auto pausePressed = Sprite::createWithSpriteFrameName("game\_pause\_pressed.png");

//CC\_CALLBACK\_1表示封装一个只有一个参数的函数，参数为（1.函数全名（含类名限定）， 2.调用函数的对象）

auto menuItemPause = MenuItemSprite::create(pauseNor, pausePressed, CC\_CALLBACK\_1(GameScene::pause, this));

menuItemPause->setAnchorPoint(Point::ANCHOR\_TOP\_RIGHT);

menuItemPause->setPosition(Point(VISIBLE\_SIZE)-Point(20, 20));

menuItemPause->setTag(111);

* 1. 实现暂停和继续函数，这里通过调用导演的pause和resume函数实现大部分功能，再新增一个类变量控制飞机移动的触摸事件是否响应

void GameScene::pause(Ref \*) {

auto menu = this->getChildByTag(110);

auto itemPause = menu->getChildByTag(111);

auto itemResume = menu->getChildByTag(112);

itemPause->setVisible(false);

itemResume->setVisible(true);

this->m\_canMove = false; //禁用触摸的变量

Director::getInstance()->pause(); //通过调用导演的pause来暂停游戏

}

listener->onTouchBegan = [=](Touch \*touch, Event \*){

…

return isContain && m\_canMove;

};

1. 添加音效
   1. 导入头文件和名称空间

#include "SimpleAudioEngine.h"

using namespace CocosDenshion;

* 1. 在WelcomeScene中预加载音乐和音效，以提高使用时的响应速度

//==============预加载声音文件

SimpleAudioEngine::getInstance()->preloadBackgroundMusic("game\_music.mp3");

SimpleAudioEngine::getInstance()->preloadEffect("achievement.mp3"); SimpleAudioEngine::getInstance()->preloadEffect("big\_spaceship\_flying.mp3");

…

* 1. 需要使用的地方进行播放

//播放背景音乐

SimpleAudioEngine::getInstance()->setBackgroundMusicVolume(0.3f);

SimpleAudioEngine::getInstance()->playBackgroundMusic("game\_music.mp3", true);

1. 处理飞机被击中后还可以移动和碰撞敌机，可以抽取一个飞机类，但这里简单起见，通过在GameScene中添加m\_planeActive来控制

//敌机与英雄的碰撞检测

if (this->m\_planeActive && enemy->getBoundingBox().intersectsRect(hero->getBoundingBox()))

{

//此变量表示飞机不再参与碰撞检测

this->m\_planeActive = false;

//此变量表示飞机不再响应触摸移动

this->m\_canMove = false;

…

}

1. 添加难度
   1. 在Enemy中添加提速变量。因为所有敌机都受同一个提速变量的影响，所以比较好的方案是使用静态变量，C++的静态变量在源文件中初始化（不是构造函数中）

//提速变量

static float m\_levelSpeed;

float Enemy::m\_levelSpeed = 0;

* 1. 为外部提供一个加速函数，用来按自己的算法实现增加提速变量的功能

void Enemy::incrementSpeed() {

//等级提升的速度（每一级提升0.2的移动距离）

if (Enemy::m\_levelSpeed ==10)

{

return;

}

Enemy::m\_levelSpeed += 0.2f;

}

* 1. 修改返回单步移动距离的函数，改为自己实现，它返回了基础移动距离和提速距离变量之和

float Enemy::getEnemyStep() {

return this->m\_enemyStep + Enemy::m\_levelSpeed;

}

* 1. 在GameScene中根据分值来提升敌机移动速度

if (enemy->getHP() <= 0)

{

//累加分钟并更新label的显示

this->m\_score += enemy->getScore();

auto lbl = (Label \*) this->getChildByTag(90);

lbl->setString(StringUtils::format("%d", this->m\_score));

**//判断是否提升等级，以提高敌机移动速度**

**if (this->m\_score%50000)**

**{**

**Enemy::incrementSpeed();**

**}**

removableEnemies.pushBack(enemy);

}

* 1. 在GameScene的init中，记得清除敌机叠加速度，否则重新游戏速度会保留上次的

void Enemy::clearLevelSpeed() {

Enemy::m\_levelSpeed = 0;

}

//注意，函数名要selector包裹，函数签名必须是void返回和float单参数

schedule(schedule\_selector(GameScene::createBullet), 0.15f);

**//敌机的等级叠加速度需要归零**

**Enemy::clearLevelSpeed();**

# 该案例笔记结束