Cocos2d-x 简单游戏元素

1. 游戏运行方式
2. Scene类、Sprite类和Action类
3. 坐标系详解
4. 内存管理机制

游戏运行方式

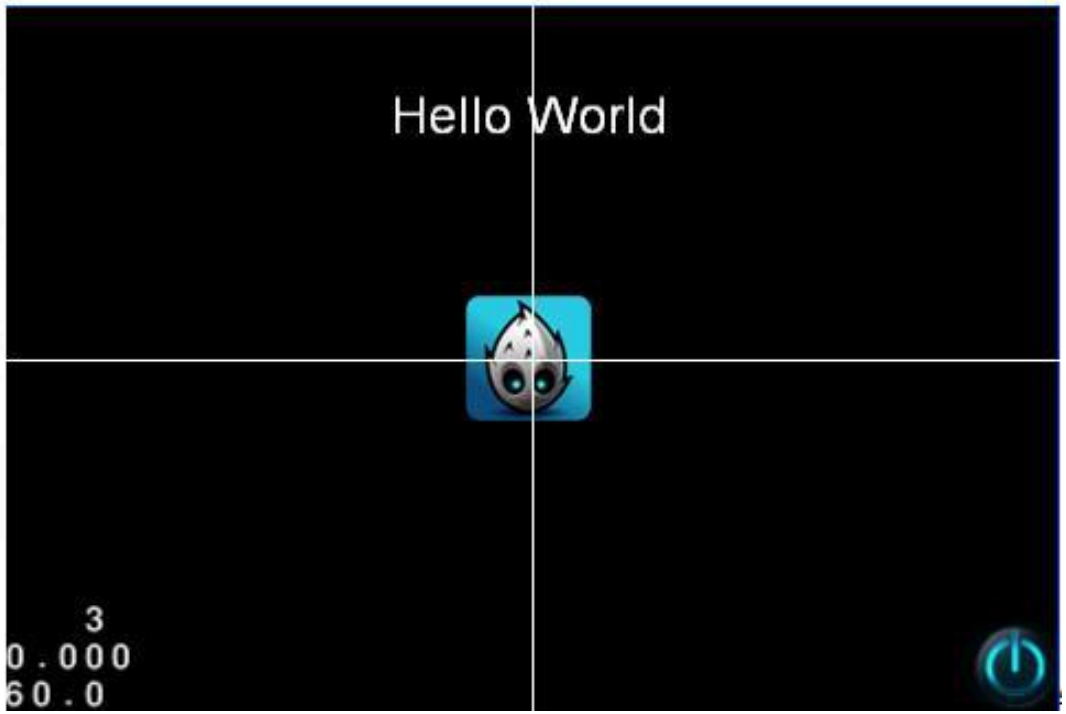
- 渲染画面
- 事件触发

游戏运行方式

渲染：

用软件从模型生成图像的过程。模型是用语言或者数据结构进行严格定义的三维物体或虚拟场景的描述，它包括几何、视点、纹理、照明和阴影等信息。图像是数字图像或者位图图像。

```
while(!exit){
    DrawGame();
}
```



游戏运行方式

事件：

事件是可以被控件识别的操作，如按下确定按钮，选择某个单选按钮或者复选框。每一种控件有自己可以识别的事件，如窗体的加载、单击、双击等事件，操控人物前进，后退，攻击等等。

```
event(){
  attack();
}
```

游戏运行方式

事件：

- 在2.x 版本事件处理时，将要触发的事件交给代理（delegate）处理，再通过实现代理里面的onTouchBegan等方法接收事件，最后完成事件的响应。

- 而在3.x的事件分发机制中，只需通过创建一个事件监听器，用来实现各种触发后的逻辑，然后添加到事件分发器_eventDispatcher，所有事件监听器由这个分发器统一管理，即可完成事件响应。

游戏运行方式

事件监听器有以下几种：

- 触摸事件 (EventListenerTouch)
- 键盘响应事件 (EventListenerKeyboard)
- 鼠标响应事件 (EventListenerMouse)
- 加速记录事件 (EventListenerAcceleration)
- 自定义事件 (EventListenerCustom)

超级马里奥

键盘操作人物的跑跳动作，通过检测碰撞，触发得分、受伤等对应事件。

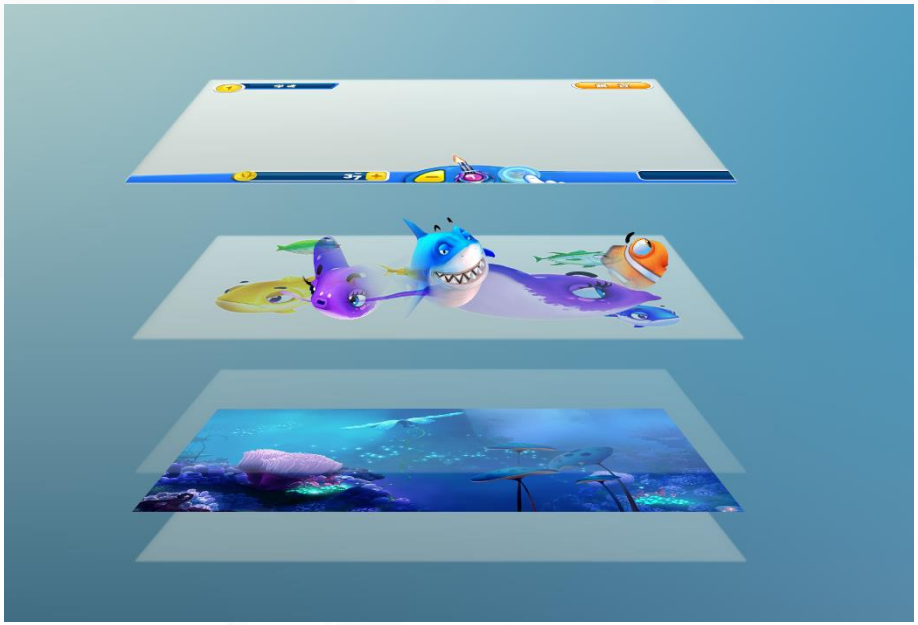
事件的触发引起游戏相应的变化，在每一帧结束时调用renderer函数，调用OpenGL代码进行图形渲染。

Scene类、Sprite类和Action类

场景切换

什么是场景？

- 场景Scene：包含游戏所需对象的容器
- 负责游戏逻辑的运行和游戏内容的逐帧渲染



几种场景切换的方式

- 单纯创建：

```
auto myScene = Scene::create();
```

- runWithScene，启动游戏第一个场景：

```
Director::getInstance()->runWithScene(myScene);
```

- replaceScene， 直接替换一个场景

```
Director::getInstance()->replaceScene(myScene);
```

几种场景切换的方式

- pushScene，暂停当前场景并入栈

```
Director::getInstance()->pushScene(myScene);
```

- popScene，推出栈中场景并运行

```
Director::getInstance()->popScene(myScene);
```

场景切换特效

- 示例

```
auto myScene = Scene::create();

// Transition Fade
Director::getInstance()->replaceScene(TransitionFade::create(0.5, myScene, Color3B(0, 255, 255)));

// FlipX
Director::getInstance()->replaceScene(TransitionFlipX::create(2, myScene));

// Transition Slide In
Director::getInstance()->replaceScene(TransitionSlideInT::create(1, myScene));
```

渐隐效果

以x轴为中心平面式旋转切换

从上向下切换

- 40种左右的场景切换特效

创建精灵

创建精灵

- PNG，JPEG，TIFF，etc. → Sprite

- 创建方式：
  1.使用指定图片创建一个Sprite

```
auto mySprite = Sprite::create("mysprite.png");
```

创建精灵

- PNG，JPEG，TIFF，etc. → Sprite

- 创建方式：
  1. 使用指定图片创建一个Sprite
  2. 使用矩形创建一个Sprite：Rect

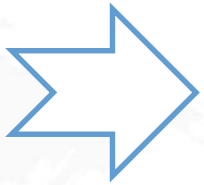
```
auto mySprite = Sprite::create("mysprite.png", Rect(0,0,40,40));
```

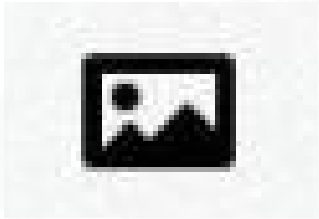
创建精灵

- PNG，JPEG，TIFF，etc. → Sprite

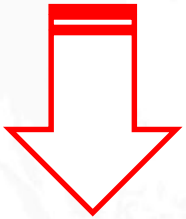
- 创建方式：
  1. 使用指定图片创建一个Sprite
  2. 使用矩形创建一个Sprite：Rect
  3. 使用Sprite Sheet创建一个精灵
     - Sprite Sheet是一个将多个精灵合并到一个文件的方法。相对于把每个精灵放在单独的文件夹中，这种方式减小了整个文件的大小。
     - 这意味着你将很大程度地减少内存的使用、文件大小和加载时间。

Sprite Sheet创建精灵

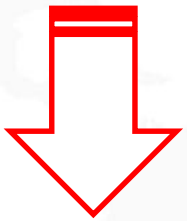


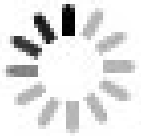


内存使用

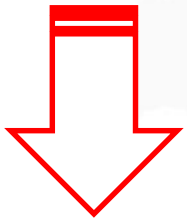


文件大小

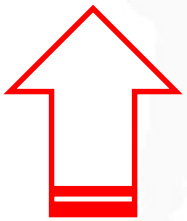




加载时间



性能



Sprite Sheet创建精灵

- 使用Sprite Sheet时

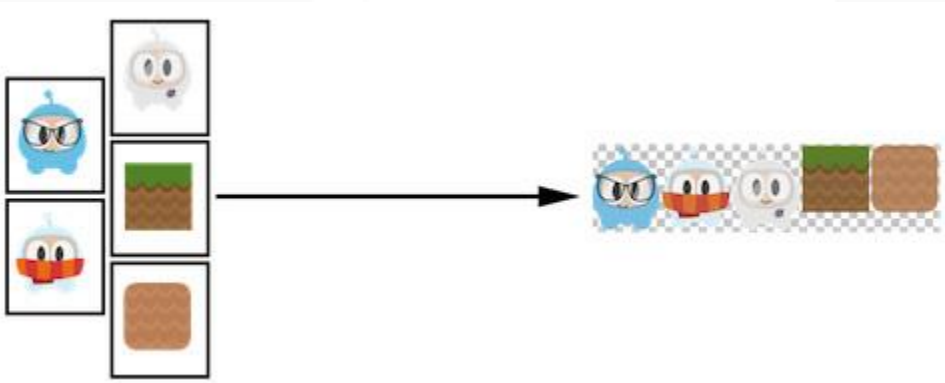
  第一步：加载到SpriteFrameCache

- SpriteFrameCache：SpriteFrame缓存类
  包含图像名和特定Sprite尺寸（Rect）

- SpriteFrameCache避免多次加载SpriteFrame

Sprite Sheet创建精灵

- Sprite Sheet 示例



Sprite Sheet创建精灵

第二步：加载一个Sprite Sheet

```
// load the Sprite Sheet
auto spritecache = SpriteFrameCache::getInstance();

// the .plist file can be generated with any of the tools mentioned below
spritecache->addSpriteFramesWithFile("sprites.plist");
```

第三步：从SpriteFrameCache中创建一个精灵

```
auto mysprite = Sprite::createWithSpriteFrameName("mysprite.png");
```

Sprite Sheet创建精灵

- 创建Sprite Sheet方法

“合图”功能

Texture Packer

Zwoptex

动作

动作

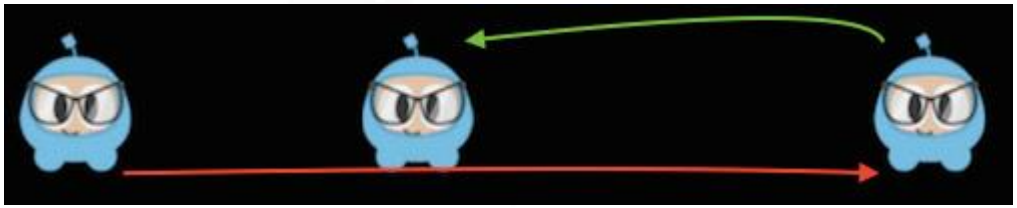
- Action类可以随时间改变Node的属性，任何一个以Node为基类的对象都有可执行的动作对象。

- 每个动作都有By和To两个状态，By相对于节点是相对的，To相对于节点是绝对的

什么是动作？

- 举个例子：

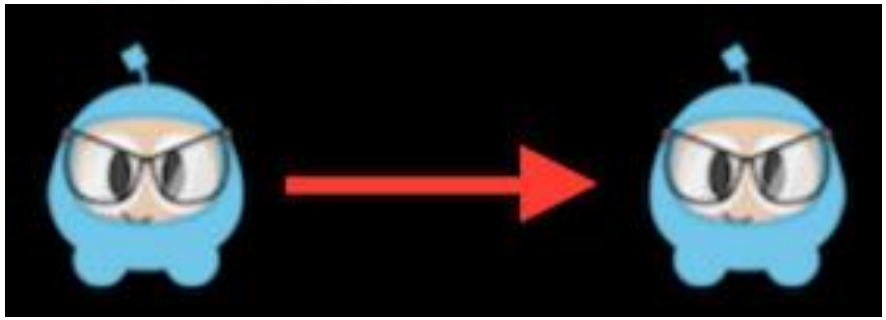
```
auto mySprite = Sprite::create("mysprite.png");
mySprite->setPosition(Vec2(200, 256));

// MoveBy - lets move the sprite by 500 on the x axis over 2 seconds
// MoveBy is relative - since x = 200 + 200 move = x is now 400 after the move
auto moveBy = MoveBy::create(2, Vec2(500, mySprite->getPositionY()));

// MoveTo - lets move the new sprite to 300 x 256 over 2 seconds
// MoveTo is absolute - The sprite gets moved to 300 x 256 regardless of
// where it is located now.
auto moveTo = MoveTo::create(2, Vec2(300, mySprite->getPositionY()));

auto delay = 2.0;

auto seq = Sequence::create(moveBy, delay, moveTo, nullptr);

mySprite->runAction(seq);
```



基本动作

- 移动（Move）

```
auto mySprite = Sprite::create("mysprite.png");

// Move a sprite to a specific location over 2 seconds.
auto moveTo = MoveTo::create(2, Vec2(50, 0));

mySprite->runAction(moveTo);

// Move a sprite 50 pixels to the right, and 0 pixels to the top over 2 seconds.
auto moveBy = MoveBy::create(2, Vec2(50, 0));

mySprite->runAction(moveBy);
```



基本动作

- 旋转（Rotate）

```
auto mySprite = Sprite::create("mysprite.png");

// Rotates a Node to the specific angle over 2 seconds
auto rotateTo = RotateTo::create(2.0f, 40.0f);
mySprite->runAction(rotateTo);

// Rotates a Node clockwise by 40 degree over 2 seconds
auto rotateBy = RotateBy::create(2.0f, 40.0f);
mySprite->runAction(rotateBy);
```

基本动作

- 缩放（Scale）

```
auto mySprite = Sprite::create("mysprite.png");

// Scale uniformly by 3x over 2 seconds
auto scaleBy = ScaleBy::create(2.0f, 3.0f);
mySprite->runAction(scaleBy);

// Scale X by 5 and Y by 3x over 2 seconds
auto scaleBy = ScaleBy::create(2.0f, 3.0f, 3.0f);
mySprite->runAction(scaleBy);

// Scale to uniformly to 3x over 2 seconds
auto scaleTo = ScaleTo::create(2.0f, 3.0f);
mySprite->runAction(scaleTo);

// Scale X to 5 and Y to 3x over 2 seconds
auto scaleTo = ScaleTo::create(2.0f, 3.0f, 3.0f);
mySprite->runAction(scaleTo);
```

基本动作

- 淡入淡出（FadeIn/FadeOut）

- Node淡入淡出，修改透明度值（0~255）

```
auto mySprite = Sprite::create("mysprite.png");

// fades in the sprite in 1 seconds
auto fadeIn = FadeIn::create(1.0f);
mySprite->runAction(fadeIn);

// fades out the sprite in 2 seconds
auto fadeOut = FadeOut::create(2.0f);
mySprite->runAction(fadeOut);
```

基本动作

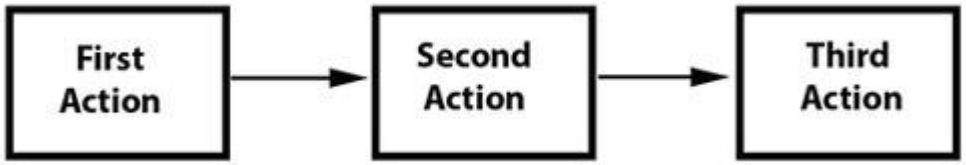
- 色调（Tint）
- 修改NodeRGB

```
auto mySprite = Sprite::create("mysprite.png");

// Tints a node to the specified RGB values
auto tintTo = TintTo::create(2.0f, 120.0f, 232.0f, 254.0f);
mySprite->runAction(tintTo);

// Tints a node BY the delta of the specified RGB values.
auto tintBy = TintBy::create(2.0f, 120.0f, 232.0f, 254.0f);
mySprite->runAction(tintBy);
```

序列动作

- 序列对象可以是，动作对象、函数（CallFunc对象）、甚至另一序列

- Sequence
- Sequence动作按添加顺序依次执行



序列动作

- 序列动作的例子

```
auto mySprite = Sprite::create("mysprite.png");

// create a few actions.
auto jump = JumpBy::create(0.5, Vec2(0, 0), 100, 1);

auto rotate = RotateTo::create(2.0f, 10);

// create a few callbacks
auto callbackJump = CallFunc::create([](){
    log("Jumped!");
});

auto callbackRotate = CallFunc::create([](){
    log("Rotated!");
});

// create a sequence with the actions and callbacks
auto seq = Sequence::create(jump, callbackJump, rotate, callbackRotate, nullptr);

// run it
mySprite->runAction(seq);
```

jump → callbackJump → rotate → callbackRotate

序列动作

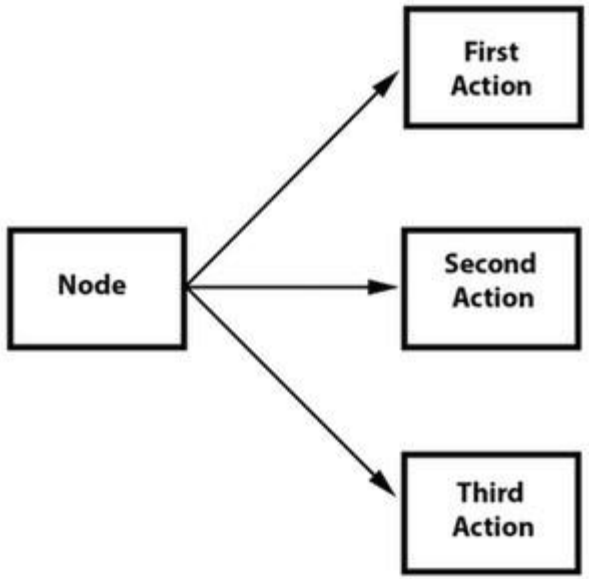
- 序列对象可以是，动作对象、函数（CallFunc对象）、甚至另一序列

- Sequence

  Sequence动作按添加顺序依次执行

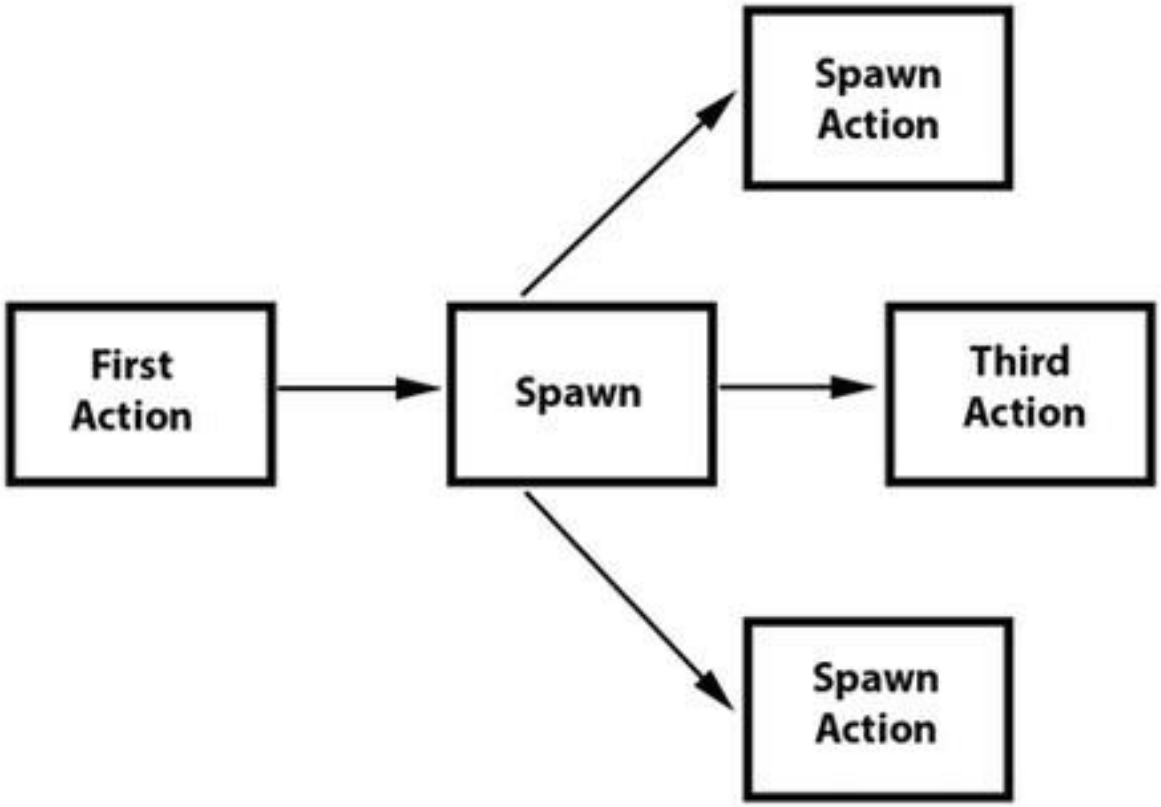
- Spawn

  Spawn所有动作同时执行



序列动作

- Spawn相似于runAction()，但可放一序列中



序列动作

- Spawn相似于runAction()，但可放一序列中

```
// create 2 actions and run a Spawn on a Sprite
auto mySprite = Sprite::create("mysprite.png");
auto moveBy = MoveBy::create(10, Vec2(400, 100));
```

- 使用Spawn：

```
// running the above Actions with Spawn.
auto mySpawn = Spawn::createWithTwoActions(moveBy, fadeTo);
mySprite->runAction(mySpawn);
```

- 使用连贯的runAction()语句：

```
// running the above Actions with consecutive runAction() statements.
mySprite->runAction(moveBy);
mySprite->runAction(fadeTo);
```

序列动作

- 序列对象可以是，动作对象、函数（CallFunc对象）、甚至另一序列

- Sequence

  Sequence动作按添加顺序依次执行
- Spawn

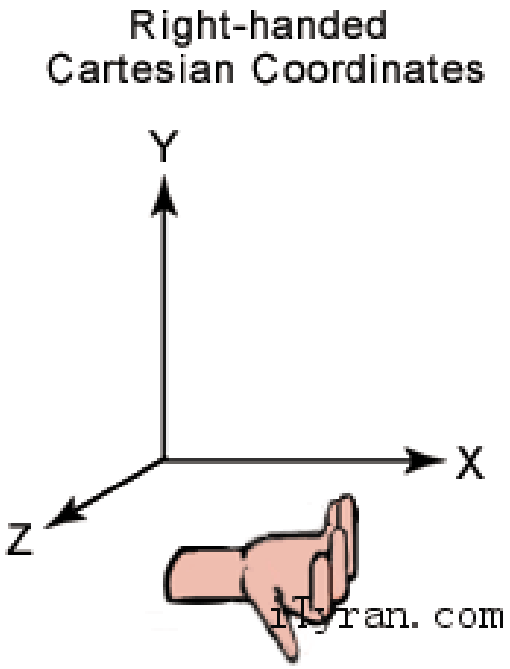
  Spawn所有动作同时执行
- 逆序（Reverse）
  - 使动作逆序执行，大部分Action和Sequence对象都是可逆的

```
// reverse a sequence, spawn or action
mySprite->runAction(mySpawn->reverse());
```

坐标系详解

笛卡尔坐标系

- 定义右手系原点在左下角，
- x向右，y向上，z向外，
- OpenGL坐标系为笛卡尔右手系



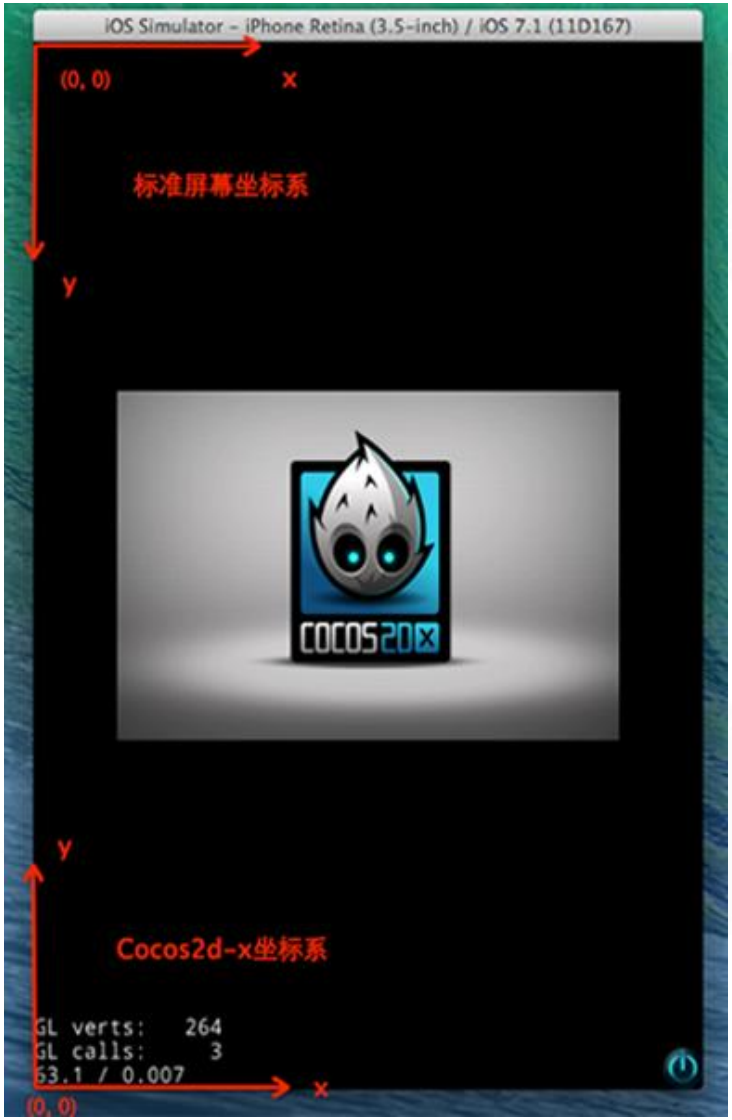
屏幕坐标系和Cocos2d坐标系

- 标准屏幕坐标系 ≠ OpenGL坐标系
- Cocos2d-x坐标系 = OpenGL坐标系

屏幕坐标系和Cocos2d坐标系

- iOS, Android, Windows Phone等在开发应用时使用的是标准屏幕坐标系，原点为屏幕左上角，x向右，y向下。

- Cocos2d坐标系和OpenGL坐标系一样，原点为屏幕左下角，x向右，y向上。



世界坐标系VS 本地坐标系

- 世界坐标系：绝对坐标系
  游戏开发中建立的概念。因此，“世界”指游戏世界。

- 本地坐标系：相对坐标系
  和节点相关联的坐标系。

- 每个节点都有独立的坐标系，当节点移动或改变方向时，和该节点关联的坐标系将随之移动或改变方向。

VertexZ，PositionZ和zOrder

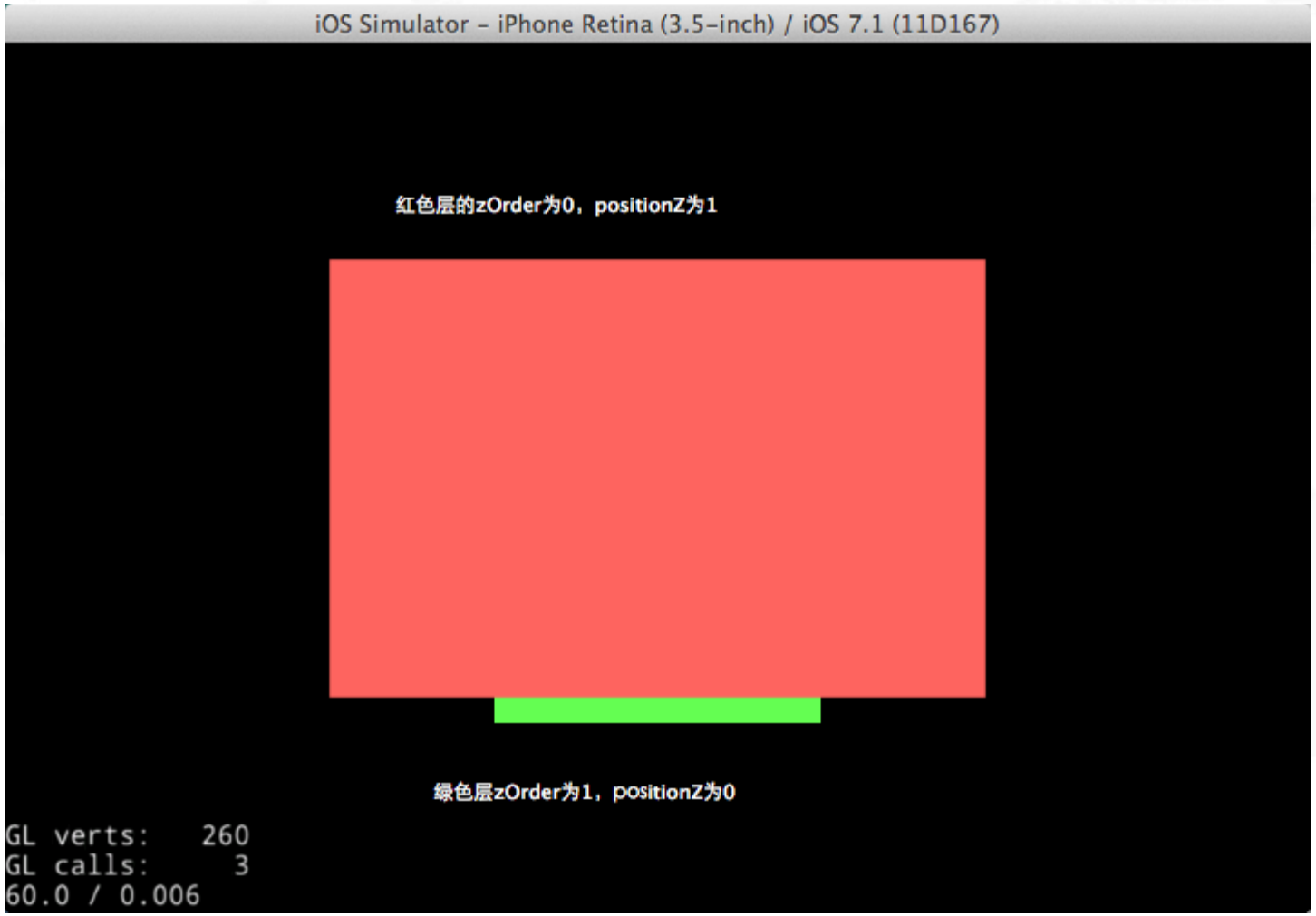
VertexZ，PositionZ和zOrder

- VertexZ是OpenGL坐标系中的Z值
- PositionZ是Cocos2d-x坐标系中Z值， 等于Vertex
- zOrder是Cocos2d-x本地坐标系中Z值

- PositionZ决定节点渲染顺序，值越大，越晚渲染
- 实际关注更多的是zOrder，zOrder则是局部渲染顺序，即该节点在其父节点上的渲染顺序，与Node的层级有关。

VertexZ，PositionZ和zOrder

- 示例说明：



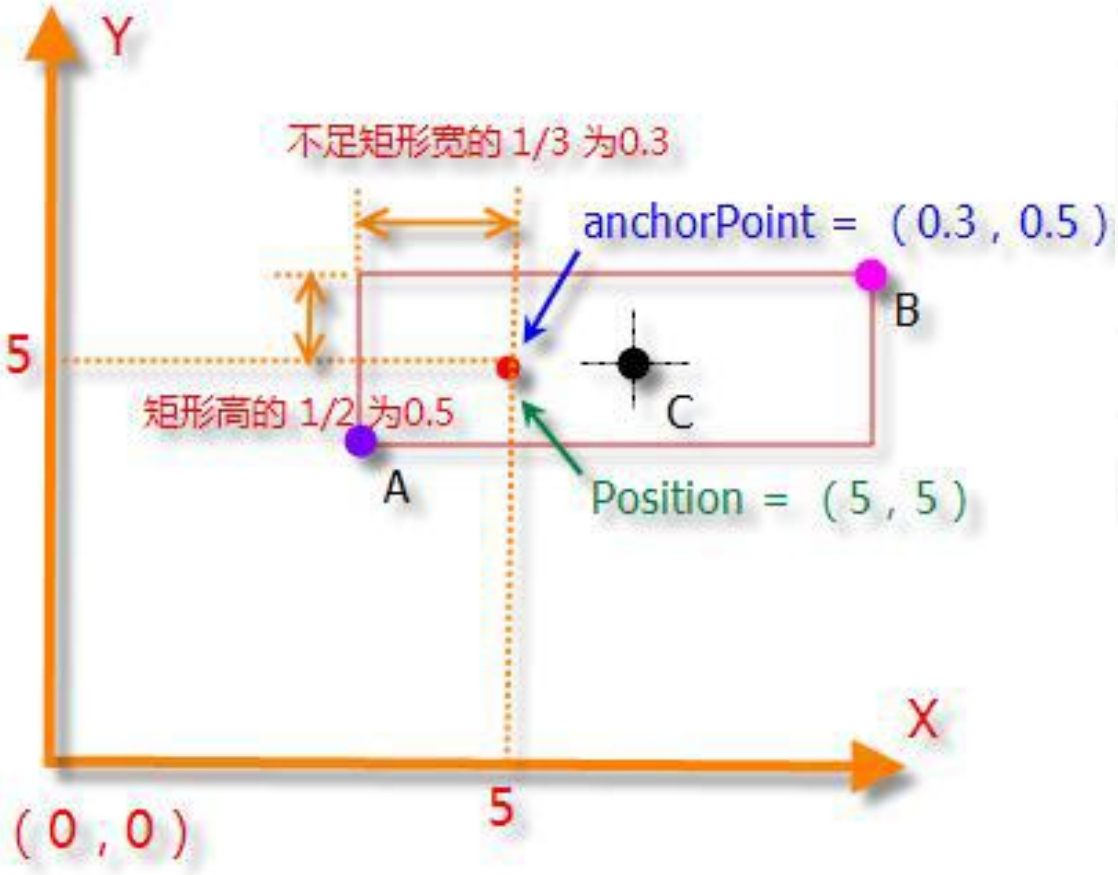
锚点Anchor

锚点Anchor

- 在cocos2d-x中，每一个对象都有锚点，父对象通过把子对象的锚点放到Position上来实现布局，对精灵的操作（例如旋转、平移），也会围绕锚点进行。

锚点Anchor

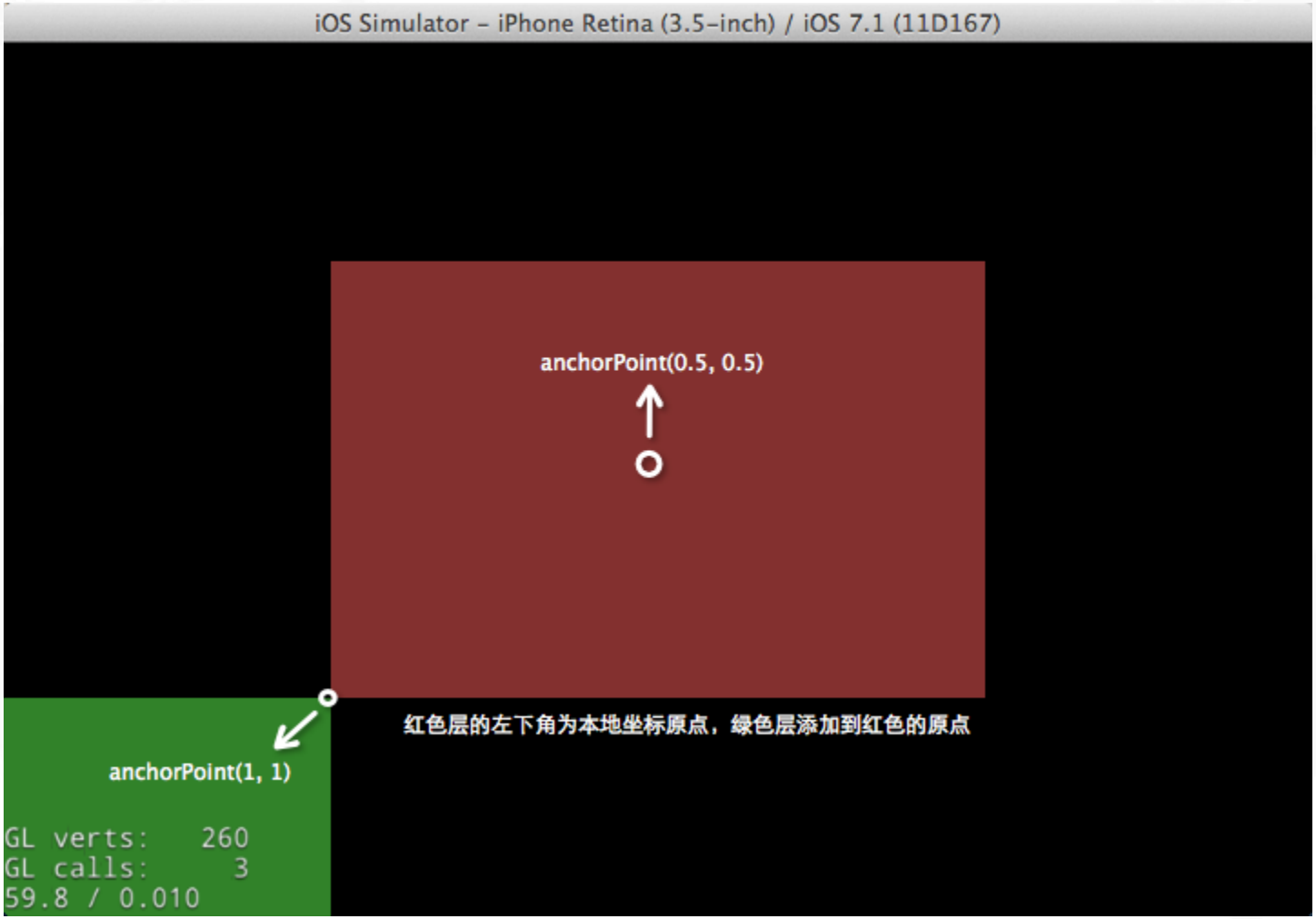
- AnchorPoint 的两个参量x 和y 的取值通常都是0 到1 之间的实数，表示锚点相对于节点长宽的位置。



锚点Anchor

- 在Cocos2d-x中Layer的Anchor Point为默认值(0, 0)，即左下角；其他Node的默认值为(0.5, 0.5)，也就是节点中心。
- 受锚点影响的精灵属性：
  - 位置Position
  - 角度Rotation
  - 缩放Scale
  - 倾斜Skew
- 不受锚点影响的精灵属性：（表面特征）
  - 颜色Color
  - 透明度Opacity

锚点Anchor


iOS Simulator - iPhone Retina (3.5-inch) / iOS 7.1 (11D167)
anchorPoint(0.5, 0.5)
红色层的左下角为本地坐标原点，绿色层添加到红色的原点
anchorPoint(1, 1)
GL verts: 260
GL calls: 3
59.8 / 0.010

锚点Anchor

- 忽略锚点(Ignore Anchor Point)
- 全称：ignoreAnchorPointForPosition，固定锚点
- Flag=true，Anchor Point固定左下角

- 示例：

```
auto red = LayerColor::create(Color4B(255, 100, 100, 128), visibleSize.width / 2, visibleSize.height / 2);
red->ignoreAnchorPointForPosition(true);
red->setPosition(Point(visibleSize.width / 2 + origin.x, visibleSize.height / 2 + origin.y));

auto green = LayerColor::create(Color4B(100, 255, 100, 128), visibleSize.width / 4, visibleSize.height / 4);
green->ignoreAnchorPointForPosition(true);

red->addChild(green);

this->addChild(red, 0);
```

锚点Anchor

- 忽略锚点(Ignore Anchor Point)

内存管理机制

内存管理机制

C++

由程序员控制，谁分配谁释放

Object-C

半自动化内存管理，谁引用谁释放，最终内存由内存回收池回收

Java

自动的内存管理方式，垃圾回收器定时回收不再使用的内存

内存管理机制

Cocos2d-x内存管理机制

虽然Cocos2d-x是用C++实现，但Cocos2d-x采用的是Object-C的内存管理方式。通过给每个对象维护一个引用计数器，记录该对象当前被引用的次数。当对象增加一次引用时，计数器加1；而对象失去一次引用时，计数器减1；当引用计数为0时，标志着该对象的生命周期结束，自动触发对象的回收释放。

内存管理机制

自动释放池（AutoReleasePool）

retain：引用计数加1；

release：引用计数减1

autorelease：告诉自动释放池，该对象对内存释放由自动释放池管理，引擎会在每帧结束之后清理autoreleasepool中的对象，调用它的release，如果对象引用计数值降为0将对象销毁。

```
Ref* Ref::autorelease()

{

    PoolManager::getInstance()->getCurrentPool()->addObject(this);        // 交给自动释放池管理

    return this;

}
```

谢谢！