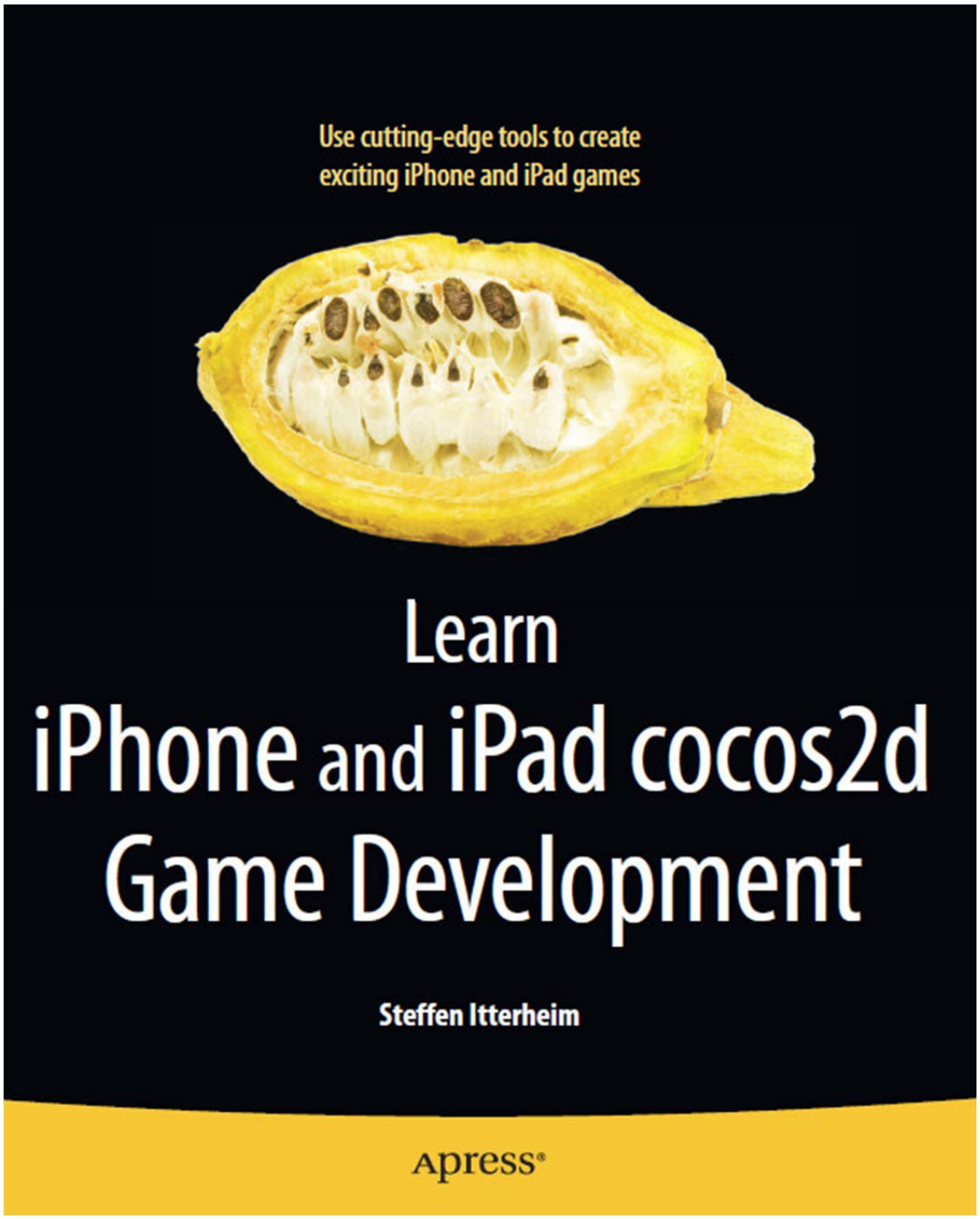

译者：杨栋
邮箱：yangdongmy@gmail.com

第十二章
物理引擎

物理引擎是很多著名的iOS游戏背后的驱动力量，比如Angry Birds（愤怒的小鸟），Stick Golf，Jelly Car和Stair Dismount。游戏开发者可以利用物理引擎创造出更动态和逼真的游戏世界。

cocos2d自带了两套物理引擎：Box2D和Chipmunk。两套引擎都是为2D游戏设计的，可以和cocos2d完美整合。

你将在本章学习两套引擎的基本知识。经过学习，你可能会发现其中一套可能更适合你。我将会简要介绍两套引擎的区别，但是最终的选择大多数是基于个人喜好的。

如果你从来都没有用过物理引擎，我在本章也会简要介绍物理引擎的基本概念和主要的组成元素。

物理引擎的基本概念

物理引擎可以被看作用于游戏物体的动画系统。当然，这取决于游戏开发者是如何将游戏物体（比如说精灵）与物理物体联系起来并且进行同步的。这里的物理物体叫做“刚体”（Rigid Bodies）。之所以叫作刚体，是因为物理引擎在驱动这些物体生成动画时，把它们当作硬的，不会变形的物体。将物体这样简化以后，物理引擎就可以同时计算大量的刚体了。

物理引擎中存在两种刚体：动态移动的（dynamic）和静态的（static）刚体。

认识到它们之间的区别很重要：静态刚体不会移动，也不应该移动-因为物理引擎可以依赖静态刚体不会互相碰撞这个特性来做出一些优化。而对于动态刚体来说，它们会相互碰撞，而且也会和静态刚体碰撞。动态刚体除了拥有位置（position）和旋转（rotation）参数，还有至少3个用于定义动态刚体的参数。它们分别是：密度或者质量（density or mass）- 用于衡量刚体有多重。另一个是摩擦力（friction）- 用于表示刚体在平面上移动时遇到阻力的大小或者有多滑。最后是复原（restitution）- 用于定义刚体的弹性。现实世界里的物体在运动时都会丢失能量，但是你可以在物理引擎中生成移动碰撞后不会丢失任何能量的动态刚体，甚至让刚体在与别的刚体发生碰撞弹回来以后，以更快的速度进行移动。

动态和静态刚体都有一个或者多个形体将刚体包含在里面。大多数情况下，这些形体是圆形或者长方形的，也可能是多边形，或者由几个顶点（vertex）组成的任意复杂的形状，或者仅仅是一条直线而已。刚体外面的这个形体是用于判断刚体之间的碰撞的。反过来，每一次碰撞会生成一些碰撞点（contact points） - 两个相碰撞刚体之间的交叉点。这些碰撞点可被用于播放粒子效果或者在刚体的碰撞处动态地添加刮痕。

物理引擎通过使用力量，脉冲和扭矩生成动态刚体的动画，而不是直接设置刚体的位置和旋转。在使用物理引擎时我不建议直接修改物体的位置和旋转信息，因为如果你手动修改物体位置信息的话，物理引擎先前所作的某些假设就会失效。

最后一点，我们可以使用关节（joints）把多个刚体连接起来，这样可以用不同的方式限制相互连接着的刚体的活动。某些关节可能配备有马达，比如它们可被用于驱动汽车的轮子或者给关节产生摩擦力，这样关节在向某个方向移动以后，会试着回到原来的位置。

物理引擎的局限性

物理引擎有它自身的局限性：它们必须在模拟效果时使用一些捷径，也就是说简化物体的复杂性，因为真实世界过于复杂，完全放到物理引擎中进行模拟是不可能的。这就是为什么要使用刚体的原因。在某些极端情况下，物理引擎有可能会捕捉不到某些已经发生的碰撞 – 例如，当刚体以很快的速度移动时，一个刚体可能直接穿透另一个刚体。虽然在量子物理学中这样的穿透情况会发生，但是我们看到的真实世界中的物体是不会相互穿透的。

刚体有时候会相互穿透卡在一起，特别是在使用了关节将它们连接在一起以后。卡在一起的刚体会努力要分开，但是为了满足关节的连接要求，它们又不得不卡在一起，结果是卡在一起的刚体会产生颤动。

我们也可能碰到游戏运行的问题。如果我们在游戏里使用了很多刚体，你永远不会知道这些刚体相互作用后的最终结果。最终，有些玩家会把自己卡死在刚体中，或者他们也可能会发现如何利用物理模拟的漏洞，跑到游戏中他们本来不应该去的区域。

物理引擎比较：Box2D 对比 Chipmunk

Cocos2d 自带了两套物理引擎:Box2D 和 Chipmunk。那么我们应该选择哪一个呢？

很多情况下，这个选择取决于个人口味。很多游戏开发者对这两个物理引擎的争论集中于它们所用的编程语言：Box2D 是用 C++写的，而 Chipmunk 用的是 C。

你可能因为 C++接口而更喜欢 Box2D。使用 C++的好处是它可以很好地与同样是面向对象的 Objective-C 进行整合。而且 Box2D 中的变量和方法名都是用全称命名的，相比之下，Chipmunk 中很多地方用的是只有一个字母的简写。再者，Box2D 中使用了运算符重载（Operator Overloading），例如，你可以将两个矢量（vector）直接相加：

```
b2Vec2 newVec = vec1 + vec2;
```

有一些功能只有Box2D提供，Chipmunk是没有的。比如，Box2D有针对快速移动物体（例如子弹）直接穿透物体而不进行碰撞测试的解决方法。

如果你不是很熟悉C++的话，你可能会发现学习C++是要花费挺多时间和精力的。这样的话，对于熟悉C语言语法或者喜欢更简单一些的物理引擎的你，可以选择Chipmunk。因为Chipmunk早Box2D几个月整合进cocos2d之中，所以网上关于Chipmunk的教程和论坛帖子也相对多一些。不过Box2D的教程数量也在赶上来。

不过有个事情要预先给你提个醒：Chipmunk使用了C的structures，它会暴露内部的域。如果你是在做试验，并且不知道某些域是拿来做什么用的，而且没有文档解释这些域的话，你就不应该改变它们 - 因为它们只被用于内部。

Chipmunk有一个很受欢迎的Objective-C接口，叫做SpaceManager。你可以利用SpaceManger很容易地把cocos2d精灵添加到刚体上，添加调试用的绘图等。你可以通过以下链接下载Chipmunk的SpaceManager：
http://code.google.com/p/chipmunkspacemanager

对于功能来说，两个选择都差不多。除非你的游戏依赖于某个物理引擎特有的功能，否则你可以使用任何一个引擎。特别是如果你对任何一个物理引擎都不熟悉，那么你可以根据引擎所使用的语言的编程风格来选择自己喜欢的引擎。

接下去，我将会为你介绍两个引擎的基础知识。你可以基于我的介绍来选择适合你的物理引擎。在接下去的一章中，你将会学习如何使用Box2D（外加VertexHelper 工具）制作一个拥有碰撞杠，翻转板和滑道的弹珠游戏台。

Box2D

Box2D是用C++写的。开发者是Erin Catto，他从2005年开始就在每一届的Game Developers Conference(GDC)上进行关于物理模拟的演讲。2007年的9月，他公开发布了Box2D引擎。从那以后，Box2D的开发工作一直很活跃。因为很受欢迎，所以cocos2d整合了Box2D与之一起发布。你可以通过在Xcode中选择File > New Project对话框中选择cocos2d Box2D模板来使用Box2D。这个工程模板会为生成的项目添加必要的Box2D源代码，并且提供一个测试项目。**图12-1**展示了这个项目。你可以在屏幕上添加盒子，然后这些盒子就会互相碰撞。根据你所拿设备的方向，这些盒子会通过重力掉落到地面上。

图12-1. PhysicsBox2D样例项目

注：因为Box2D使用C++写的，所以你必须使用.mm作为你所有项目的实现文件的后缀名，而不是通常的.m后缀名。.mm后缀用于告知Xcode把有此后缀名的文件作为Objective-C++或者C++代码来处理。如果你使用了.m后缀，Xcode就会把代码以Objective-C和C来处理，这样Xcode就不能正确处理Box2D的C++代码了。因此，如果你碰到很多编译错误的话，请首先检查一下是不是所有的实现文件都是以.mm作为后缀的。

Box2D的文档存在于两个地方。首先，你可以在以下网址阅读Box2D的手册：http://www.box2d.org/manual.html,这个手册会介绍基本的概念和样例代码。其次，Box2D的API参考可以在此下载：http://code.google.com/p/box2d。你也可以在本书的源代码中找到Box2D的API参考。你可以在以下文件夹中找到：*/Physics Engine Libraries/Box2D_v2.1.2/Box2D/Documentation/API*，找到并点击文件夹中的index.html来打开参考文档。

如果你喜欢Box2D，你可以考虑向这个项目捐款。你可以通过官网（http://www.box2d.org）上的Donate按钮进行捐款。

由Box2D创造的世界

由于cocos2d提供的例子比较复杂，我在这里将其分解成比较小几个部分，然后一步一步地重建这个项目。当然，我也在重建的过程中添加了一些额外的东西和变化。

列表12-1展示了PhysicsBox2D01项目中的 HelloWorldScene 头文件：

列表12-1. Box2D Hello World头文件

```
#import "cocos2d.h"
#import "Box2D.h"
#import "GLES-Render.h"
enum
{
```

```
        kTagBatchNode,
};

@interface HelloWorld : CCLayer
{
        b2World* world;
}
+(id) scene;
@end
```

这是个很标准的头文件，除了我们添加的Box2D.h头文件和一个b2World类型的成员变量。这个变量就是我们要床罩的物理世界 - 你可以把它想像成一个容器类，用于存放和更新所有的物理刚体。

我们通过 HelloWorldScene中的init方法进行b2World对象的初始化，如**列表12-2**所示：

列表12-2. 对Box2D世界进行初始化

```
b2Vec2 gravity = b2Vec2(0.0f, -10.0f);
bool allowBodiesToSleep = true;
world = new b2World(gravity, allowBodiesToSleep);
```

Box2D是用C++写的，所以在创建新的Box2D类的指针时，你必须在类名前使用 new 这个关键词。如果Box2D是Objective-C写的话，上述的代码可能会像下面的样子：

```
world = [[b2World alloc] initWithGravity:gravity allowSleep:allowBodiesToSleep];
```

换句话说，C++中的new关键词和Objective-C中的alloc关键词拥有同样的功能。创建了新的对象也意味着必须在使用完成后释放对象，在C++中使用delete关键词进行内存释放的：

```
-(void) dealloc
{
        delete world;
        [super dealloc];
}
```

列表12-2中生成的Box2D世界初始化时使用了一个初始的重力矢量值和用于决定是否允许动态刚体“睡眠”的标记变量。

会“睡眠”的动态刚体是什么意思呢？让刚体“睡眠”是在物理模拟中使用的一个技巧，它允许系统在进行模拟时快速跳过不需要处理的刚体。当施加到某个刚体上的力量小于临界值一段时间以后，这个刚体将会进入“睡眠”状态。换句话说，如果某个刚体移动或者旋转的很慢或者根本不在动的话，物理引擎将会把它标记为“睡眠”状态，不再对其施加力量 – 除非新的力量施加到刚体上让其再次移动或者旋转。通过把一些刚体标记为“睡眠”状态，物理引擎可

以省下很多时间。除非你游戏中的所有动态刚体处于持续的运动中，你应该把allowBodiesToSleep变量设置为true，如**列表12-2**所示。

传入Box2D世界的重力是一个b2Vec2的struct类型。它和CGPoint在本质上是一样的，因为它们都储存着x轴和y轴的浮点值。在我们的例子里，和真实世界一样，重力都是一个常量。(0, -10)这个矢量持续不变地施加到所有动态刚体上，让它们坠落，在我们的例子里就是往屏幕底部坠落。

把活动范围限制在屏幕之内

Box2D世界已经设置完成，接下去我们要把Box2D中刚体的活动范围限制在可视屏幕范围之内。要达到这个目的，我们需要一个静态刚体。最简单的就是通过使用world的CreateBody方法和一个空的刚体定义来生成一个静态刚体：

```
//定义静态刚体容器用于屏幕边缘的碰撞测试
b2BodyDef containerBodyDef;
b2Body* containerBody = world->CreateBody(&containerBodyDef);
```

刚体通常都是使用world的CreateBody方法来生成的，这样可以确保正确地分配和释放刚体所占用的内存。这里的b2BodyDef是一个struct，用来存放所有用于生成刚体的数据，比如位置和刚体类型。默认情况下，一个空的刚体定义会生成一个位于(0,0)位置的静态刚体。

注：&containerBodyDef变量前面的&符号在C++中的意思是：“请给我containerBodyDef的内存地址”。如果看一下CreateBody方法的定义，你会看到它要求传入一个指针(pointer)：b2World::CreateBody(const b2BodyDef *def);.指针存放着变量的内存地址。你可以通过在非指针类型变量名前面加上&符号让其返回此变量的指针。

刚体本身不会做任何事情。要让刚体把屏幕区域包起来，你必须创建一个有四个边的刚体形状：

```
//通过给一个多边形单独分配四个边来生成包含屏幕的盒子
b2PolygonShape screenBoxShape;
int density = 0;

// 底部
screenBoxShape.SetAsEdge(lowerLeftCorner, lowerRightCorner);
containerBody->CreateFixture(&screenBoxShape, density);

// 顶部
screenBoxShape.SetAsEdge(upperLeftCorner, upperRightCorner);
containerBody->CreateFixture(&screenBoxShape, density);

// 左边
screenBoxShape.SetAsEdge(upperLeftCorner, lowerLeftCorner);
containerBody->CreateFixture(&screenBoxShape, density);
```

```
// 右边
screenBoxShape.SetAsEdge(upperRightCorner, lowerRightCorner);
containerBody->CreateFixture(&screenBoxShape, density);
```

你可能注意到了用于储存各个边角坐标值的变量名 - upperLeftCorner，lowerRightCorner等。我将很快解释这些变量。首先，我想让你关注b2PolygonShape screenBoxShape这个变量是如何被重用的。每一次调用SetAsEdge方法以后，我都会调用 containerBody->CreateFixture()，并且传入 &screenBoxShape作为参数。这里我使用了 ->运算符，是因为containerBody是一个C++指针。因为Box2D生成了一个screenBoxShape的拷贝，所以你可以安全地重用同一个形状以生成包含屏幕区域的四个边，而不需要更新或者覆盖之前的代码。因为这里的刚体是静态刚体，所以它们的密度可以被设为0.

注：因为b2PolygonShape类有SetAsBox方法，所以看起来好像我们可以直接把屏幕的宽和高传给这个方法以得到包含屏幕区域的刚体。不过如果你这样做的话，实际上生成的是一个实心的刚体。任何添加到屏幕上的动态刚体将会被包含在一个实心刚体中，而不是我们需要的空心刚体。结果是添加到屏幕上的动态刚体将会与实心刚体发生碰撞，从而以很快的速度离开屏幕。因此，屏幕的四个边需要单独创建，以便只有四个边是实心的，中间则是空心的，用于放入其它刚体。

现在我们讨论之前没有解释过的那些边角变量。请注意以下代码中，屏幕的宽度和高度值除以一个名为 PTM_RATIO的常量，把像素值转换成了以米为单位来计算长度：

```
//需要用到的一些数值
CGSize screenSize = [CCDirector sharedDirector].winSize;
float widthInMeters = screenSize.width / PTM_RATIO;
float heightInMeters = screenSize.height / PTM_RATIO;
b2Vec2 lowerLeftCorner = b2Vec2(0, 0);
b2Vec2 lowerRightCorner = b2Vec2(widthInMeters, 0);
b2Vec2 upperLeftCorner = b2Vec2(0, heightInMeters);
b2Vec2 upperRightCorner = b2Vec2(widthInMeters, heightInMeters);
```

为什么我们要用米来作为长度的单位呢？PTM_RATIO又是什么？Box2D在0.1米到10米的范围内工作是最优化的，因为它针对这个范围做过专门优化。对Box2D来说，所有的距离是以米为单位的，所有的物体都是以公斤为单位的，时间则是以秒为单位的。如果你对米，公斤和秒（MKS）系统不熟悉的话，不要担心 - 你没有必要把码数（yards）转换成米，把磅转换成公斤，因为在Box2D中把长度以米为单位是要把长度范围设定在0.1米和10米之间，而刚体的重量本来就和真实世界中的重量没有多大关系。对于刚体的重量，我们在调试的时候更多的是依靠感觉，而不是实际的重量值。

你应该把创建的Box2D世界中的刚体的大小限定在越接近1米越好。不过这并不

意味着你不能有长度小于0.1米的刚体，或者长度大于10米的刚体。但是，太小或者太大的刚体很可能会在游戏运行过程中产生错误和奇怪的行为。

PTM_RATIO的定义如下：

```
#define PTM_RATIO 32
```

PTM_RATIO用于定义32个像素在Box2D世界中等同于1米。一个有32像素宽和高的盒子形状的刚体等同于1米宽和高的物体。在Box2D中，4x4像素的大小是0.125x0.125。你可以通过PTM_RATIO把刚体的尺寸设置成最适合Box2D的尺寸，而PTM_RATIO设置为32，对于拥有1024x768像素的iPad来说也是很合适的。

在b2Vec2和CGPoint之间转换

请注意：b2Vec2和CGPoint是不同的，这意味着你不能够在需要CGPoint的地方使用b2Vec2，反过来也不行。而且，Box2D里的点需要转换成米为单位，或者从米为单位转换回以像素为单位。为了避免出错，比如忘记转换单位，或者打错了字，或者把x轴坐标使用了两次，我强烈建议你把这些重复的转换代码封装到方法中去：

```
-(b2Vec2) toMeters:(CGPoint)point
{
	return b2Vec2(point.x / PTM_RATIO, point.y / PTM_RATIO);
}

-(CGPoint) toPixels:(b2Vec2)vec
{
	return ccpMult(CGPointMake(vec.x, vec.y), PTM_RATIO);
}
```

上述方法可以让你在CGPoint和b2Vec2之间的转换变得更容易：

```
CGPoint point = CGPointMake(100, 100);
b2Vec2 vec = b2Vec2(200, 200);

CGPoint pointFromVec;
pointFromVec = [self toPixels:vec];

b2Vec2 vecFromPoint;
vecFromPoint = [self toMeters:point];
```

在Box2D世界中添加盒子

现在，屏幕的四边都设置了静态的刚体，它们所包含的屏幕区域中唯一缺少的是一些动态的刚体。让我们来放些盒子形状的刚体进去。

我把 David Gervais 的90度角瓷砖集(dg_grounds32.png)添加到了PhysicsBox2D01项目的Resources文件夹中。这些瓷砖都是32x32像素大小的，

所以它们的尺寸刚好是1x1米大小。列表12-3展示了init方法中用于添加贴图和创建盒子的代码。这段代码也预约了更新方法，用于更新盒子精灵的位置信息。我们也在这里启用了触摸功能，这样用户就可以通过用手指点击屏幕生成一个新的盒子了。

列表12-3. 通过添加一些盒子状刚体进行初始化

```
//使用90度角瓷砖集来生成小盒子
CCSpriteBatchNode* batch = [CCSpriteBatchNode batchNodeWithFile:@"dg_grounds32.png"
                                                        capacity:150];
[self addChild:batch z:0 tag:kTagBatchNode];

// 添加几个初始的刚体
for (int i = 0; i < 11; i++)
{
        [self addNewSpriteAt:CGPointMake(screenSize.width / 2, screenSize.height / 2)];
}

[self scheduleUpdate];
self.isTouchEnabled = YES;
```

列表12-4中的 addNewSpriteAt方法是cocos2d Box2D应用模板项目中的一部分。不过我稍稍改动了一下，让它可以利用瓷砖集里的所有瓷砖：

列表12-4. 在点击的位置生成一个由精灵作为表现的动态刚体

```
-(void) addNewSpriteAt:(CGPoint)pos
{
        CCSpriteBatchNode* batch = (CCSpriteBatchNode*)[self getChildByTag:kTagBatchNode];
        int idx = CCRANDOM_0_1() * TILESET_COLUMNS;
        int idy = CCRANDOM_0_1() * TILESET_ROWS;
        CGRect tileRect = CGRectMake(TILESIZE * idx, TILESIZE * idy, TILESIZE, TILESIZE);
        CCSprite* sprite = [CCSprite spriteWithBatchNode:batch rect:tileRect];
        sprite.position = pos;
        [batch addChild:sprite];

        //创建一个刚体的定义，并将其设置为动态刚体
        b2BodyDef  bodyDef;
        bodyDef.type = b2_dynamicBody;
        bodyDef.position = [self toMeters:pos];
        bodyDef.userData = sprite;
        b2Body* body = world->CreateBody(&bodyDef);

        // 定义一个盒子形状，并将其复制给body fixture
        b2PolygonShape  dynamicBox;
        float tileInMeters = TILESIZE / PTM_RATIO;
        dynamicBox.SetAsBox(tileInMeters * 0.5f, tileInMeters * 0.5f);
```

```
        b2FixtureDef fixtureDef;
        fixtureDef.shape = &dynamicBox;
        fixtureDef.density = 0.3f;
        fixtureDef.friction = 0.5f;
        fixtureDef.restitution = 0.6f;
        body->CreateFixture(&fixtureDef);
}
```

首先，通过使用CCSprite的spriteWithBatchNode初始化方法，并向其提供一个32x32像素大小的CGRect，从CCSpriteBatchNode生成一个精灵。我在瓷砖集中随机地取出一块瓷砖作为精灵的贴图。

然后，生成一个刚体。但是，这次我们将b2BodyDef的type属性设置为b2_dynamicBody，这让生成的刚体成为了动态刚体，可以到处移动，也能与别的动态刚体发生碰撞。之前生成的精灵则被赋值给了userData属性。之后，当你遍历world中的刚体时，你可以利用userData属性快速访问刚体的精灵。

刚体的形状是b2PolygongShape，设置成了0.5米大小的盒子形状。因为使用SetAsBox方法生成的盒子是其接受的参数大小的两倍，所以提供给这个方法的参数坐标值要除以2 - 或者像我一样乘以0.5f。这样得到的最终盒子的大小是1米宽和高的。

最后，生成的动态刚体需要一个fixture - 可以把fixture理解成包含着刚体需要用到的所有数据的容器。这些数据包括：刚体的形状（最重要的一项），密度，摩擦力和复原（这项会影响刚体在world中移动和弹跳的方式）。

把精灵和刚体连接起来

精灵不会自动跟着刚体移动和旋转。如果你没有定期地调用Box2D world中的Step方法的话，刚体也不会做任何事情。在调用Step方法后，你就需要得到刚体的位置和角度信息，然后赋值给精灵，以便更新精灵的位置信息。更新精灵的步骤发生在如**列表12-5**所示的更新方法中：

列表12-5. 更新每个刚体相关联的精灵的位置和旋转信息

```
-(void) update:(ccTime)delta
{
        //使用固定的时间间隔将物理模拟向前推进一步
        float timeStep = 0.03f;
        int32 velocityIterations = 8;
        int32 positionIterations = 1;
        world->Step(timeStep, velocityIterations, positionIterations);

        for (b2Body* body = world->GetBodyList(); body != nil; body = body->GetNext())
        {
```

```
                CCSprite* sprite = (CCSprite*)body->GetUserData();
                if (sprite!= NULL)
                {
                        sprite.position = [self toPixels:body->GetPosition()];
                        float angle = body->GetAngle();
                        sprite.rotation = CC_RADIANS_TO_DEGREES(angle) * -1;
                }
        }
}
```

Box2D的world是通过定期地调用Step方法来实现动画的。Step方法需要三个参数。第一个是timeStep，它会告诉Box2D自从上次更新以后已经过去多长时间了，直接影响着刚体会在这一步移动多长距离。对于游戏来说，我不建议你使用delta time来作为timeStep的值，因为delta time会上下浮动，这样的话刚体就不能以相同的速度移动了。比如，当你的设备在后台用了十分之一秒的时间进行收发邮件的操作。这会让你的刚体在下一帧移动很长一段距离。在这种情况下，对于游戏来说，你宁可让游戏停止十分之一秒，也比让游戏继续进行要好。如果没有一个固定的timeStep值，物理引擎会通过让刚体基于不同的时间来移动，以达到符合短暂中断的操作。如果时间间隔之间的差距很大，刚体会在某些帧中做大距离的移动。

Step方法的第二和第三个参数是迭代次数。它们被用于决定物理模拟的精确程度，也决定着计算刚体移动所需要的时间。这是对于速度和准确度的取舍。对于位置的迭代次数（positionIterations），一次就够了。速度的迭代次数（velocityIterations）更加重要；一个比较好的速度迭代次数值是8。在游戏中，超过10次的迭代不会带来明显的作用，带式1到4此迭代是无法得到稳定的模拟结果的。速度的迭代次数越少，刚体的行为就越起伏不定。我想最好的方法是对这些值进行一些试验，以得到自己想要的效果。

在world向前推进一步以后，接下去的for循环会使用 world->GetBodyList和body->GetNext方法，对world中的所有刚体进行遍历。在遍历每个刚体时，我把刚体的userData返回，并且转换成CCSprite指针。如果精灵存在，刚体的位置信息就会被转换成像素值，并赋值给精灵的位置属性，让精灵可以随着刚体一起移动。同样的，我也得到了刚体的角度值，因为这个值是以弧度为单位的，所以我把它用cocos2d的CC_RADIANS_TO_DEGRESS方法将其转换成度，然后乘以-1，这样精灵就可以用刚体一样的方向进行旋转了。

碰撞测试

Box2D有一个名为 b2ContactListener的类。如果你想接受碰撞的回调，你应该创建一个继承自b2ContactListener的新类。以下代码请参考 PhysicsBox2D02项目。

在Xcode中创建一个新类，命名为ContactListener。然后将实现文件改名为ContactListenenr.mm。这样你就可以在同一个文件中使用C++和Objective-C代

码了。列表12-6展示了ContactListener类的头文件：

列表12-6：ContactListener类的头文件

```
#import "Box2D.h"

class ContactListener : public b2ContactListener
{
	private:
		void BeginContact(b2Contact* contact);
		void EndContact(b2Contact* contact);
};
```

这是一个C++类，因此这个类的定义有些不一样。请注意最后那个闭合大括号后面的分号。很多人会忘记那个分号。BeginContact和EndContact这两个方法是Box2D定义的，任何时候两个刚体发生碰撞时都会调用这两个方法。

在实现代码中，当两个刚体发生碰撞时，我只是简单地把精灵的颜色换成了洋红色。然后在两个刚体分开以后将它们的颜色换回白色。代码如列表12-7所示：

列表12-7.检测碰撞的开始和结束

```
#import "ContactListener.h"
#import "cocos2d.h"
void ContactListener::BeginContact(b2Contact* contact)
{
	b2Body* bodyA = contact->GetFixtureA()->GetBody();
	b2Body* bodyB = contact->GetFixtureB()->GetBody();
	CCSprite* spriteA = (CCSprite*)bodyA->GetUserData();
	CCSprite* spriteB = (CCSprite*)bodyB->GetUserData();

	if (spriteA != NULL && spriteB != NULL)
	{
		spriteA.color = ccMAGENTA;
		spriteB.color = ccMAGENTA;
	}
}

void ContactListener::EndContact(b2Contact* contact)
{
	b2Body* bodyA = contact->GetFixtureA()->GetBody();
	b2Body* bodyB = contact->GetFixtureB()->GetBody();
	CCSprite* spriteA = (CCSprite*)bodyA->GetUserData();
	CCSprite* spriteB = (CCSprite*)bodyB->GetUserData();

	if (spriteA != NULL && spriteB != NULL)
	{
		spriteA.color = ccWHITE;
```

```
                spriteB.color = ccWHITE;
        }
}
```

b2Contact包含了所有与碰撞相关的信息。这些信息包含了代码中以A和B为后缀的两个相碰撞的刚体的所有信息。这里没有区别是谁碰撞了谁 - 这两个刚体发生了相互碰撞。例如，你的游戏中的敌人被玩家的子弹击中了，你当然是让敌人受伤，而不是子弹受伤。谁碰撞了谁和对谁造成了什么样的伤害是由你来决定的。还有一点要注意：碰撞测试方法可能会在一帧里面被调用多次，每一次碰撞都会调用一次这个方法。

虽然通过contact，进入fixture，然后取得刚体（body），最后用刚体的GetUserData方法得到精灵这一整个过程很复杂，但是Box2D的API参考可以帮助你理解我们在这里所经过的层级。理解以后，通过这样的方式获取精灵将会变成你的第二天性。

要使用ContactListener，你必须把它添加到world中。在HelloWorldScene中导入 ContactListener.h头文件，然后在这个类中添加 ContactListener* contactListener 作为一个成员变量：

```
#import "cocos2d.h"
#import "Box2D.h"
#import "GLES-Render.h"
#import "ContactListener.h"
...
@interface HelloWorld : CCLayer
{
        b2World* world;
        ContactListener* contactListener;
}
...
@end
```

现在，在HelloWorldScene的init方法中，你可以创建一个新的ContactListner实例，然后将它设为 world的contact listener：

```
contactListener = new ContactListener();
world->SetContactListener(contactListener);
```

剩下的事情就是在dealloc方法中添加用于删除 contactListener的代码了：

```
-(void) dealloc
{
        delete contactListener;
        delete world;
        [super dealloc];
}
```

现在，PhysicsBox2D02项目中的盒子在碰到其他盒子时会变成紫色了。

用关节把刚体连接起来

我们可以用关节(joint)把刚体连接起来。我们所使用的关节类型决定着刚体连接的方式。在以下的例子中，我生成了4个刚体。其中3个是动态刚体，我使用了“旋转关节”(revolute joint)来连接它们。这种关节会让刚体之间保持相同的距离，并且刚体可以360度旋转。如果你想像不出来这些刚体会如何运动，你应该编译运行PhysicsBox2D02项目。第四个刚体是一个静态刚体，它同其中一个动态刚体也是用“旋转关节”相连的：

```
-(void) addSomeJoinedBodies:(CGPoint)pos
{
        //生成一个刚体定义，将其设置为动态刚体
        b2BodyDef bodyDef;
        bodyDef.type = b2_dynamicBody;

        // 位置信息必须被转换成以米为单位
        bodyDef.position = [self toMeters:pos];
        bodyDef.position = bodyDef.position + b2Vec2(-1, -1);
        bodyDef.userData = [self addRandomSpriteAt:pos];
        b2Body* bodyA = world->CreateBody(&bodyDef);
        [self bodyCreateFixture:bodyA];

        bodyDef.position = [self toMeters:pos];
        bodyDef.userData = [self addRandomSpriteAt:pos];
        b2Body* bodyB = world->CreateBody(&bodyDef);
        [self bodyCreateFixture:bodyB];

        bodyDef.position = [self toMeters:pos];
        bodyDef.position = bodyDef.position + b2Vec2(1, 1);
        bodyDef.userData = [self addRandomSpriteAt:pos];
        b2Body* bodyC = world->CreateBody(&bodyDef);
        [self bodyCreateFixture:bodyC];

        // 生成旋转关节
        b2RevoluteJointDef jointDef;
        jointDef.Initialize(bodyA, bodyB, bodyB->GetWorldCenter());
        bodyA->GetWorld()->CreateJoint(&jointDef);

        jointDef.Initialize(bodyB, bodyC, bodyC->GetWorldCenter());
        bodyA->GetWorld()->CreateJoint(&jointDef);

        // 生成一个隐形的静态刚体，并把body A连接到它上面
        bodyDef.type = b2_staticBody;
        bodyDef.position = [self toMeters:pos];
```

```
        b2Body* staticBody = world->CreateBody(&bodyDef);

        jointDef.Initialize(staticBody, bodyA, bodyA->GetWorldCenter());
        bodyA->GetWorld()->CreateJoint(&jointDef);
}
```

在生成刚体时我重用了b2BodyDef，然后为每个刚体设置了不同的位置信息和使用了随机生成的CCSprite赋值给刚体的userData。如前所述，addRandomSpriteAt这个方法包含着利用CCSpriteBatchNode生成精灵的代码。因为addSomeJoinedBodies这个方法里需要好几个精灵，所以把生成精灵的代码放到addRandomSpriteAt方法中是很有好处的。

b2RevoluteJointDef的初始化方法需要三个参数：头两个是需要相连接起来的刚体，第三个参数是关节本身的坐标。通过使用其中一个刚体的GetWorldCenter方法获取的坐标值，那个刚体就会被放在关节的中心点，并且这个刚体只能围绕自己旋转。

关节是通过使用b2World类的CreateJoint方法来生成的。虽然PhysicsBox2D02项目中的HelloWorldScene类里有一个b2World的成员变量，但是我想在这里演示一下其实你可以通过任何刚体来得到b2World - 不管通过哪个刚体，你只要使用刚体的 GetWorld 方法就可以得到b2World。这是个很有用的方法。因为在之前讨论过的ContactListener中，并不存在b2World成员变量。因此，你必须通过刚体的这个方法来获取b2World。

Chipmunk

Chipmunk物理引擎有Howling Moon Software的Scott Lembcke开发。Chipmunk实际上受到了早期版本的Box2D的启发。你可以通过Google Code网站下载它的代码。如果你喜欢Chipmunk，你可以考虑向这个项目捐款。你可以通过以下网站的Donate按钮进行捐款：http://code.google.com/p/chipmunk-physics

Chipmunk的文档位于Scott的网站上：
http://files.slembcke.net/chipmunk/release/ChipmunkLatest-Docs。

如果你需要帮助，你可以使用Chipmunk的论坛：www.slembcke.net/forums.

面向对象的Chipmunk

实际上存在两套Chipmunk的Objective-C封装，而且还有更多在开发中的封装。我不会在本书讨论这些封装，但是你应该知道有这些封装存在，并且找时间试用一下。

Scott的Objective-C封装是Chipmunk代码的一部分，不过它只能在iPhone模拟器中工作。如果你要把它发布到真实设备中的话，你必须在Howling Moon

Software网站上购买Scott的Objective-C封装代码：
http://howlingmoonsoftware.com/objectiveChipmunk.php.

你最好试用了以后再买。你可以通过以下链接中的教程试用一下如何使用Scott的Objective-C封装在iPhone模拟器上运行：
http://files.slembcke.net/chipmunk/tutorials/SimpleObjectiveChipmunk.

另一个选择是Chipmunk SpaceManager，由Robert Blackwood开发，它是免费的。不过，它的主要目的是可以让cocos2d更容易地整合Chipmunk。如果你要试用SpaceManager，你可以通过以下链接下载（你也可以通过相同页面进行捐款）：
http://code.google.com/p/chipmunk-spacemanager.

SpaceManager的API参考文档可以在Robert的网站上找到：
www.mobilebros.com/spacemanager/docs.

两套Chipmunk的Objective-C封装代码，和它们的文档都包含在本章的源代码文件夹中，位于Physics Engine Libraries子文件夹中。

创建Chipmunk的物理世界

接下去我将用Chipmunk来创建一个与之前一样的物理模拟世界。我将从PhysicsChipmunk01项目开始讨论，并且讨论如何为Chipmunk物理引擎进行初始化设置。**列表12-8**展示了 HelloWorldScene头文件：

列表12-8. Chipmunk的HelloWorldScene头文件

```
#import "cocos2d.h"
#import "chipmunk.h"
enum
{
        kTagBatchNode = 1,
};

@interface HelloWorld : CCLayer
{
        cpSpace* space;
}
+(id) scene;
@end
```

上述代码没有什么特别的地方，除了cpSpace这个成员变量。在Chipmunk中，它所模拟的物理世界不叫world，而是叫space。虽然用词不一样，但是它们的实际意义是完全一样的。Chipmunk的space包含了所有的刚体。

Chipmunk通过 HelloWorldScene的init方法来进行初始化，初始化代码如下：

```
cpInitChipmunk();
```

```
space = cpSpaceNew();
space->iterations = 8;
space->gravity = CGPointMake(0, -100);
```

在使用任何Chipmunk方法之前，第一件必须做的事情是调用 cpInitChipmunk方法。然后，你可以使用 cpSpaceNew来生成space，并且设置space的迭代次数 - 在我们的例子中我将其设置为8。这个迭代次数和我在Box2D例子里的update方法中用到的值是一样的。Chipmunk只有一种迭代 - elasticIterations这项已经过时不用了。如果你熟悉Chipmunk，你需要注意到这一点。如果你的游戏不需要刚体可以叠在一起，你可以用小于8的迭代次数；否则，你会发现叠在一起的刚体要经过很长时间才会停止颤动和滑动，最终停下来。

你可能注意到Chipmunk可以使用应用于iPhone SDK中的CGPoint结构。Chipmunk内部使用的结构叫做 cpVect，但是在cocos2d中你可以使用任何一个。我在这里使用了一个CGPoint将重力设为-100 - 这个数值所产生的重力将会和Box2D项目中用到的重力大致相同。

当然，我们需要将space进行内存释放。这是通过调用 cpSpaceFree方法，将space作为参数传入方法来实现的：

```
-(void) dealloc
{
	cpSpaceFree(space);
	[super dealloc];
}
```

生成包含屏幕的静态刚体

要把所有的盒子限制在屏幕的4边之内，我们需要生成一个静态刚体，用刚体的四个边把屏幕区域包含在里面。首先，我们定义屏幕四个角的变量：

```
CGSize screenSize = [CCDirector sharedDirector].winSize;
CGPoint lowerLeftCorner = CGPointMake(0, 0);
CGPoint lowerRightCorner = CGPointMake(screenSize.width, 0);
CGPoint upperLeftCorner = CGPointMake(0, screenSize.height);
CGPoint upperRightCorner = CGPointMake(screenSize.width, screenSize.height);
```

和Box2D不同，你不需要把像素转换成以米为单位。你可以直接使用屏幕的像素尺寸来定义四个角，你也可以在Chipmunk的刚体上使用像素为单位。

接着，我将通过使用 cpBodyNew来生成静态刚体。此方法接受的两个参数都被设置为INFINITY，这会让生成的刚体成为一个静态刚体。这两个参数是：质量（mass）和惯性（inertia），将这两个参数设为INFINITY以后，生成的刚体将不会移动。

```
// 生成一个静态刚体，用它把屏幕包含其中以把其他刚体限制在屏幕之内
float mass = INFINITY;
float inertia = INFINITY;
```

```
cpBody* staticBody = cpBodyNew(mass, inertia);
```

注：Chipmunk中的质量(Mass)和惯性(Inertia)是和Box2D中的密度(Density)和摩擦力(Friction)相对应的。惯性和摩擦力的区别是：前者决定着刚体开始移动时的阻力，后者决定着当刚体与别的刚体发生碰撞时会丢失的动能。

接着，如**列表12-9**所示，我们将定义刚体的形状（shape），这个形状组成了屏幕边界：

列表12-9. 生成屏幕边界

```
cpShape* shape;
float elasticity = 1.0f;
float friction = 1.0f;
float radius = 0.0f;

// 底部
shape = cpSegmentShapeNew(staticBody, lowerLeftCorner, lowerRightCorner, radius);
shape->e = elasticity;
shape->u = friction;
cpSpaceAddStaticShape(space, shape);

// 顶部
shape = cpSegmentShapeNew(staticBody, upperLeftCorner, upperRightCorner, radius);
shape->e = elasticity;
shape->u = friction;
cpSpaceAddStaticShape(space, shape);

// 左边界
shape = cpSegmentShapeNew(staticBody, lowerLeftCorner, upperLeftCorner, radius);
shape->e = elasticity;
shape->u = friction;
cpSpaceAddStaticShape(space, shape);

// 右边界
shape = cpSegmentShapeNew(staticBody, lowerRightCorner, upperRightCorner, radius);
shape->e = elasticity;
shape->u = friction;
cpSpaceAddStaticShape(space, shape);
```

cpSegmentShapeNew方法用于生成4个新线段，用于定义屏幕的4个边。为了方便起见，shape变量在这里被重复利用，但是shape变量要求在每次调用cpSegmentShapeNew方法以后都要设置弹性值(elasticity)（这和Box2D中的回复力(restitution)是一样的）和摩擦力(friction)。然后，我们通过cpSpaceAddStaticShape方法将每个shape作为静态刚体添加到space中去。

注：在Chipmunk中，你不得不经常用到只有一个字母的字段，比如e和u。以我

个人之见，我想这是使用Chipmunk的一个难点，因为你很难一下子明白这些一个字母的字段到底代表了什么，结果是你不得不时常地查看Chipmunk文档。

添加盒子

我在HelloWorldScene的init方法中使用了和Box2D例子里一样的代码，向模拟世界添加盒子。你可以查看**列表12-3**中的代码以刷新你的记忆。

我将直接在这里讨论如何生成新的动态刚体盒子，如**列表12-10**中的addNewSpriteAt方法所示：

列表12-10. 以Chipmunk的方式添加附带精灵的刚体到space中

```
-(void) addNewSpriteAt:(CGPoint)pos
{
	float mass = 0.5f;
	float moment = cpMomentForBox(mass, TILESIZE, TILESIZE);
	cpBody* body = cpBodyNew(mass, moment);

	body->p = pos;
	cpSpaceAddBody(space, body);

	float halfTileSize = TILESIZE * 0.5f;
	int numVertices = 4;
	CGPoint vertices[] =
	{
		CGPointMake(-halfTileSize, -halfTileSize),
		CGPointMake(-halfTileSize, halfTileSize),
		CGPointMake(halfTileSize, halfTileSize),
		CGPointMake(halfTileSize, -halfTileSize),
	};

	CGPoint offset = CGPointZero;
	float elasticity = 0.3f;
	float friction = 0.7f;

	cpShape* shape = cpPolyShapeNew(body, numVertices, vertices, offset);
	shape->e = elasticity;
	shape->u = friction;
	shape->data = [self addRandomSpriteAt:pos];
	cpSpaceAddShape(space, shape);
}
```

我们通过使用cpBodyNew方法来生成代表盒子的动态刚体，这个方法需要两个参数：质量（mass）和惯性力矩（moment of inertia）。惯性力矩决定着刚体移动时遇到的阻力，它是通过 cpMomentForBox这个帮助方法（helper method）

来计算的。cpMomentForBox以刚体的质量和盒子的尺寸作为参数。在我们的例子里，盒子的尺寸就是瓷砖的尺寸，也就是32x32像素大小。

然后我们通过cpSpaceAddBody方法来更新刚体的位置信息(p)和把刚体添加到space中。请注意：和Box2D不同，你不需要在Chipmunk里把像素转换成米；你可以直接使用像素坐标。

接着，我创建了一系列的顶点（Vertex），这些顶点将会作为定义盒子的四个角。因为盒子的四个角的位置是相对于盒子的中心点来放置的，所以它们离开中心点的位置都是瓷砖尺寸的一半大小。否则，盒子将会变成两倍于瓷砖的大小。

我们将生成的刚体，顶点数组，顶点数组的顶点数量和一个可选的偏移值（offset）（在我们的例子里设成了CGPointZero），传给cpPolyShapeNew方法。得到的结果是盒子形状（shape）的cpShape指针。获取的形状指针拥有和Box2D例子中的盒子类似的弹性和摩擦力属性。然后，精灵被赋值给这个指针的data属性，接着我们通过cpSpaceAddShape方法把shape指针添加到space中。

列表12-11中的 addRandomSpriteAt方法只是简单地生成了一个CCSprite对象，这个对象会随着动态刚体的移动而移动。

列表12-11. 利用随机图片生成新的盒子对象

```
-(CCSprite*) addRandomSpriteAt:(CGPoint)pos
{
	CCSpriteBatchNode* batch = (CCSpriteBatchNode*)[self getChildByTag:kTagBatchNode];

	int idx = CCRANDOM_0_1() * TILESET_COLUMNS;
	int idy = CCRANDOM_0_1() * TILESET_ROWS;

	CGRect tileRect = CGRectMake(TILESIZE * idx, TILESIZE * idy, TILESIZE, TILESIZE);
	CCSprite* sprite = [CCSprite spriteWithBatchNode:batch rect:tileRect];
	sprite.position = pos;
	[batch addChild:sprite];

	return sprite;
}
```

更新盒子的精灵

和Box2D一样，你必须每一帧都更新精灵的位置和旋转信息来使它和动态刚体的位置和旋转同步。这需要在update方法中进行实现：

```
-(void) update:(ccTime)delta
{
	float timeStep = 0.03f;
	cpSpaceStep(space, timeStep);
```

```
        // 调用 forEachShape这个C方法以更新精灵的位置
        cpSpaceHashEach(space->activeShapes, &forEachShape, nil);
        cpSpaceHashEach(space->staticShapes, &forEachShape, nil);
}
```

和Box2D一样，你必须使用step方法推进物理模拟。在Chipmunk中，我们需要使用cpSpaceStep方法，此方法以space和timeStep作为参数。我们使用的timeStep是固定的数值，因为如果使用delta时间的话会使物理模拟变得不稳定。

我们通过使用cpSpaceHashEach方法来调用forEachShape这个C 方法以访问每一个shape。或者，说的准确一点，访问每一个动态刚体，然后是每一个静态刚体。对于forEachShape方法的第三个参数，你可以传入任意的指针 - 因为我们的例子不需要第三个参数，所以我把它设为nil。虽然本例没有属于静态刚体的精灵，但是我还是把对静态刚体的方法调用放在了这里，以防万一你以后想添加静态刚体进去。

forEachShape是一个用C写的回调方法。在本例中，你可以在HelloWorldScene.m文件中找到它，位于@implementation代码外面。虽然没有严格规定必须放在 @implementation代码外面，但是这样做可以让代码更清晰，表明这个方法不是HelloWorldScene类的一部分。这个方法被定义为static，这让它成为一个全局的方法。代码如**列表12-12**所示：

列表12-12. 更新刚体精灵的位置和旋转信息

```
//用于更新精灵位置和旋转信息的C方法:
static void forEachShape(void* shapePointer, void* data)
{
        cpShape* shape = (cpShape*)shapePointer;
        CCSprite* sprite = (CCSprite*)shape->data;
        if (sprite != nil)
        {
                cpBody* body = shape->body;
                sprite.position = body->p;
                sprite.rotation = CC_RADIANS_TO_DEGREES(body->a) * -1;
        }
}
```

传给 cpSpaceHashEach方法的参数被严格地定义为两个，而且都必须是void类型的指针。第二个参数就是传给cpSpaceHashEach方法的第三个参数。对于这两个参数，你必须确认它们指向的对象是什么类型的；否则你有可能会遇到EXC_BAD_ACCESS这样的错误。

在本例中，我知道shapePointer指向的是一个cpShape结构，因此我可以安全地把它转换成cpShape类型，然后将shape的data域设置为CCSprite指针。如果我们使用的精灵指针是有效的，我可以从shape中获取到刚体，然后将精灵的位置

和旋转信息设为和刚体一样。和Box2D的例子一样，旋转信息必须从弧度转换成度，然后乘以-1以得到正确的旋转方向。

Chipmunk的碰撞测试

Chipmunk的碰撞测试也是由C写的回调方法来处理的。在PhysicsChipmunk02项目中，我添加了 cotactBegin和contactEnd这两个静态方法（如**列表12-13**所示），它们的功能和Box2D中的一样：会把发生碰撞的盒子颜色变成洋红色。

列表12-13. Chipmunk的碰撞测试回调方法

```
static int contactBegin(cpArbiter* arbiter, struct cpSpace* space, void* data)
{
	bool processCollision = YES;

	cpShape* shapeA;
	cpShape* shapeB;
	cpArbiterGetShapes(arbiter, &shapeA, &shapeB);

	CCSprite* spriteA = (CCSprite*)shapeA->data;
	CCSprite* spriteB = (CCSprite*)shapeB->data;
	if (spriteA != nil && spriteB != nil)
	{
		spriteA.color = ccMAGENTA;
		spriteB.color = ccMAGENTA;
	}

	return processCollision;
}

static void contactEnd(cpArbiter* arbiter, cpSpace* space, void* data)
{
	cpShape* shapeA;
	cpShape* shapeB;
	cpArbiterGetShapes(arbiter, &shapeA, &shapeB);

	CCSprite* spriteA = (CCSprite*)shapeA->data;
	CCSprite* spriteB = (CCSprite*)shapeB->data;
	if (spriteA != nil && spriteB != nil)
	{
		spriteA.color = ccWHITE;
		spriteB.color = ccWHITE;
	}
}
```

如果测得碰撞并且方法正常运行的话，contactBegin会返回YES。如果返回的是

NO 或者0，你可以忽略碰撞。为了得到精灵，首先你必须从cpArbiter得到shape。CpArbiter就像b2Contact一样，包含着碰撞各方的信息。通过cpArbiterGetShapes方法，把两个shape传给它作为赋值只用，你会得到发生碰撞的两个刚体shapeA和shapeB。然后，你可以通过shapeA和shapeB获取它们各自的CCSprite指针。如果两个精灵指针都有效的话，它们的颜色将会发生变化。

和Box2D一样，这些回调方法不会被自动调用。在 HelloWorldScene的init方法中，在space生成以后，你必须使用 cpSpaceAddCollisionHandler方法把上述两个回调方法添加进space里：

```
unsigned int defaultCollisionType = 0;
cpSpaceAddCollisionHandler(space, defaultCollisionType, defaultCollisionType,
                                                    &contactBegin, NULL, NULL,
                                                    &contactEnd, NULL);
```

默认的shape碰撞测试类型是0。因为本例中我不关心碰撞的类型究竟是什么，所以我把两个碰撞类型的参数都设为0。你可以为每个刚体的shape添加一个整型（integer）作为它的collision_type属性，然后把碰撞回调方法添加到space，这样的话只有碰撞双方的碰撞属性相同时，碰撞才会发生。这叫作“碰撞过滤”（filtering collisions），在Chipmunk的手册里有描述：http://files.slembcke.net/chipmunk/release/ChipmunkLatest-Docs/#cpShape.

接下去的四个参数是对应四个碰撞阶段的C写的回调方法名：开始（begin），预处理（pre-solve），后处理（post-solve）和分开（separation）（最后一步和Box2D中的EndContact事件一样）。这些和Box2D中相对应的回调方法是一样的。大多数时候，你只需要开始和分开时间相对应的回调方法就足够了。

我在预处理和后处理的地方传入了NULL作为参数，因为我没有兴趣处理这两个事件。你可以使用这两个方法来影响碰撞或者在后处理阶段获取碰撞力量。最后一个参数是一个任意数据指针（data pointer），你可以将它传入回调方法中去。我不需要传任何数据到回调方法中，所以在这里设置了NULL。

这样，你就完成了碰撞测试回调方法的设置。

Chipmunk中的关节

Chipmunk例子也需要自己的 addSomeJoinedBodies方法的实现。这里的实现代码比Box2D要来的罗嗦，如**列表12-14**所示。你会任何大多数设置静态和动态刚体的代码。你可以直接跳到最后关于生成关节的代码。

列表12-14. 生成三个用关节连接起来的刚体

```
-(void) addSomeJoinedBodies:(CGPoint)pos
{
        float mass = 1.0f;
        float moment = cpMomentForBox(mass, TILESIZE, TILESIZE);
```

```
float halfTileSize = TILESIZE * 0.5f;
int numVertices = 4;
CGPoint vertices[] =
{
        CGPointMake(-halfTileSize, -halfTileSize),
        CGPointMake(-halfTileSize, halfTileSize),
        CGPointMake(halfTileSize, halfTileSize),
        CGPointMake(halfTileSize, -halfTileSize),
};

//生成静态刚体
cpBody* staticBody = cpBodyNew(INFINITY, INFINITY);
staticBody->p = pos;

CGPoint offset = CGPointZero;
cpShape* shape = cpPolyShapeNew(staticBody, numVertices, vertices, offset);
cpSpaceAddStaticShape(space, shape);

// 生成三个新的动态刚体
float posOffset = 1.4f;
pos.x += TILESIZE * posOffset;
cpBody* bodyA = cpBodyNew(mass, moment);
bodyA->p = pos;
cpSpaceAddBody(space, bodyA);

shape = cpPolyShapeNew(bodyA, numVertices, vertices, offset);
shape->data = [self addRandomSpriteAt:pos];
cpSpaceAddShape(space, shape);

pos.x += TILESIZE * posOffset;
cpBody* bodyB = cpBodyNew(mass, moment);
bodyB->p = pos;
cpSpaceAddBody(space, bodyB);

shape = cpPolyShapeNew(bodyB, numVertices, vertices, offset);
shape->data = [self addRandomSpriteAt:pos];
cpSpaceAddShape(space, shape);

pos.x += TILESIZE * posOffset;
cpBody* bodyC = cpBodyNew(mass, moment);

bodyC->p = pos;
cpSpaceAddBody(space, bodyC);
```

```
        shape = cpPolyShapeNew(bodyC, numVertices, vertices, offset);
        shape->data = [self addRandomSpriteAt:pos];
        cpSpaceAddShape(space, shape);

        // 生成关节，并且给空间添加约束
        cpConstraint* constraint1 = cpPivotJointNew(staticBody, bodyA, staticBody->p);
        cpConstraint* constraint2 = cpPivotJointNew(bodyA, bodyB, bodyA->p);
        cpConstraint* constraint3 = cpPivotJointNew(bodyB, bodyC, bodyB->p);

        cpSpaceAddConstraint(space, constraint1);
        cpSpaceAddConstraint(space, constraint2);
        cpSpaceAddConstraint(space, constraint3);
}
```

在上述例子终哦你，我用 cpPivotJointNew方法生成了一个支点（pivot point），这和在Box2D的例子中使用 b2RevoluteJoint方法是一样的效果。我们通过使用两个互相连接的刚体和其中一个刚体的中心位置信息（作为定位点-anchor point）。cpPivotJointNew方法将会返回一个cpConstraint指针，你必须使用 cpSpaceAddConstraint方法将其添加到space中。

结语

本章我们学习了cocos2d自带的两个物理引擎Box2D和Chipmunk的基本知识。现在你有了针对这两个引擎的两个可以工作的例子代码，它们可以帮助你决定你想用哪个物理引擎。

你学习了如何设置屏幕区域，让这个区域可以包含所有使用瓷砖集生成的动态刚体。你现在也知道如何进行碰撞测试和如何生成用于连接刚体的关节。

下一章，你将使用Box2D物理引擎制作一个游戏。