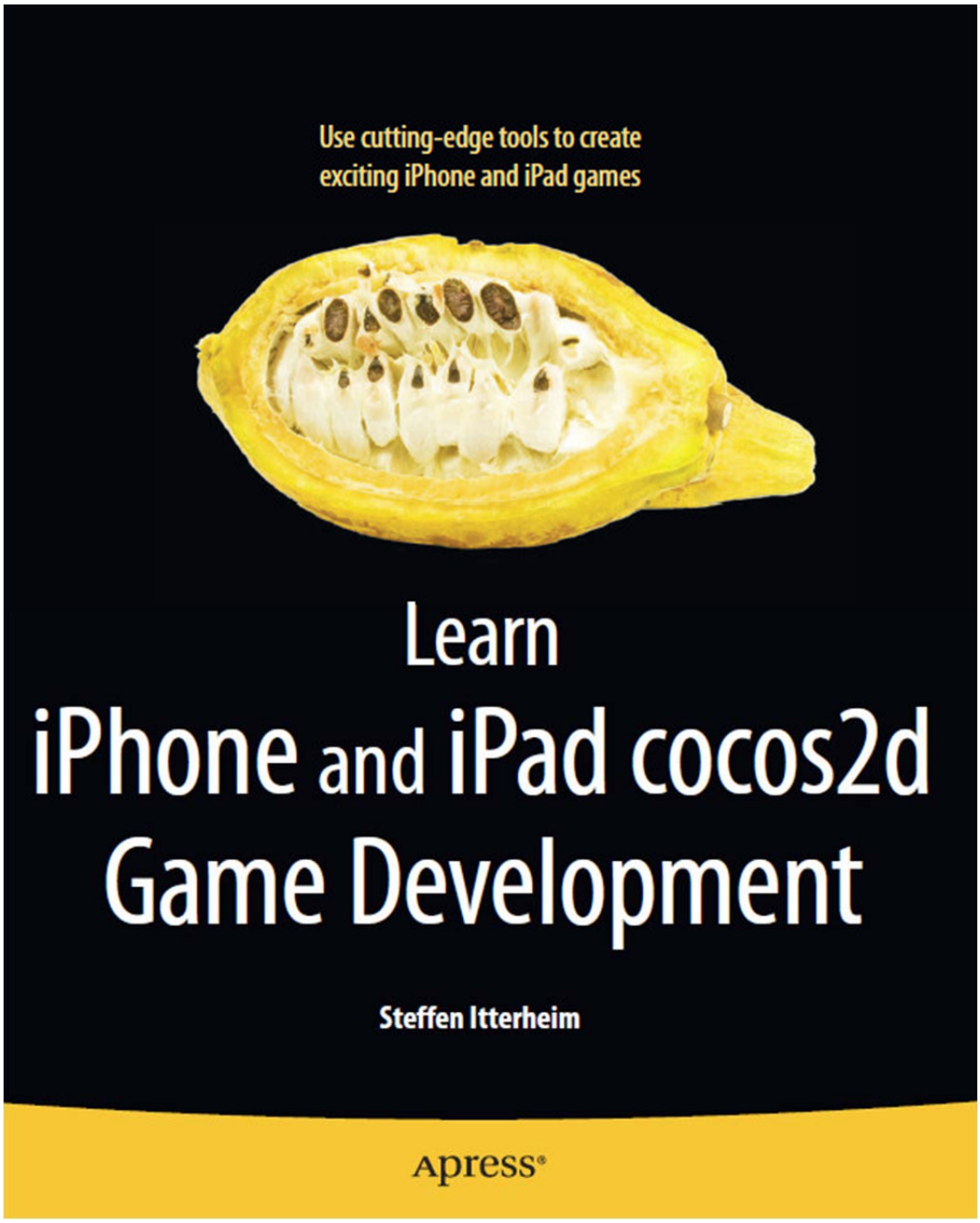

译者：杨栋
邮箱：yangdongmy@gmail.com

第十三章
弹球游戏

我将在本章使用Box2D物理引擎制作一款弹球游戏。弹球游戏实际上就是利用了物理世界的特性，将其转化为好玩的游戏。不过，对于物理引擎来说，你不仅仅只局限于真实世界的物理特性。

弹球桌上的一些元素，比如碰撞器和球，可以通过为它们选择相应的摩擦力，弹性和密度来创建。其他一些元素则需要使用关节才能工作 – 对于翻板（Flipper），我们需要使用旋转关节（revolute joint）；对于活塞（Plunger），我们需要使用柱状关节（prismatic joint）。当然，你还需要很多用于定义桌面碰撞测试用的静态刚体。

因为在源代码里定义碰撞测试用的多边形是不现实的 - 如果要制作一款拥有一定细节的令人置信的弹球游戏。我将在这里介绍另一个有用的工具：VertexHelper。你可以使用这个工具，通过把顶点一个接一个画出来的方式来生成碰撞测试用的多边形。

在本章结束的时候，你将拥有一个可以玩的弹球游戏，如图13-1所示：

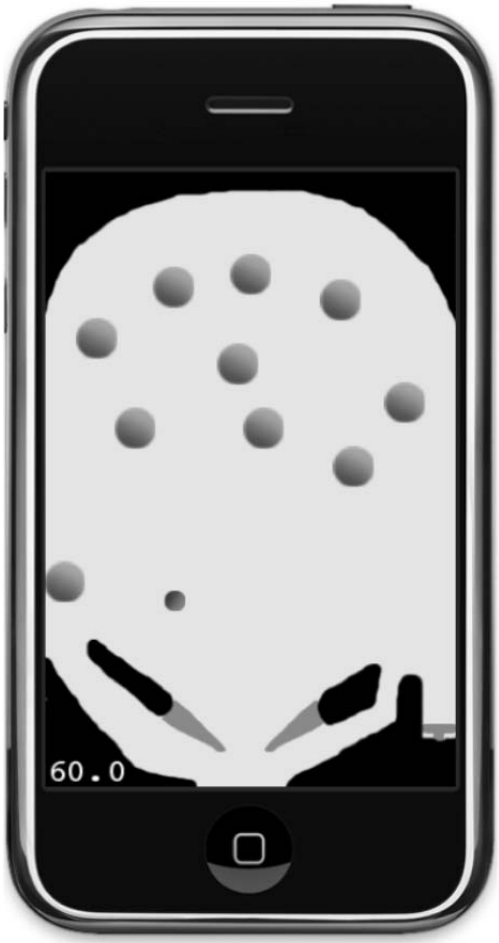

图13-1. 制作完成的弹球游戏

形状(Shapes):凸面体(Convex)和反时针方向

让我们从对碰撞测试用的多边形的要求开始讨论。首先，当你为Box2D和Chimunk定义碰撞多边形时，你要认识到这些引擎有以下两个要求：

1. 组成多边形的顶点在定义是要以反时针方向来进行。
2. 多边形必须是凸面体。

凸面体的特点是：你可以在凸面体里面找任意两个点，这两个点连成的线的任何部分都不会落在凸面体外面。凹面体（Concave）刚好和凸面体相反：凹面体里任意两个点连成的线可能有一部分会落在外面。图13-2展示了凸面体和凹面体之间的区别。

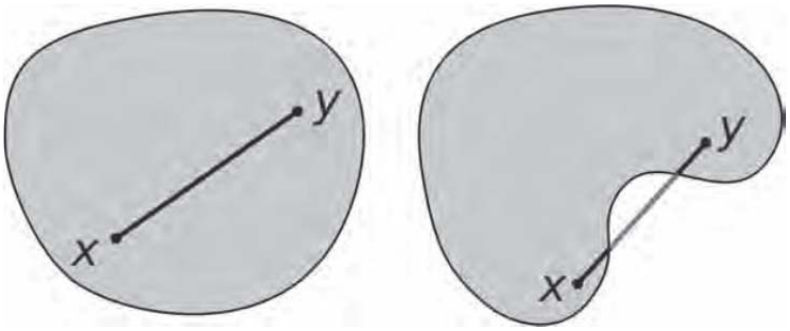

图13-2. 凸面体和凹面体之间的区别

如果你不清楚如何以反时针方向定义凸面体顶点的话，你可以先在脑子里想像一下：首先在任意一个地方放一个顶点，然后在第一个顶点的左边画第二个。接着在左下方画第三个，右下方画第四个，再连上第一个顶点，这样你就用反时针方向的方式画出了一个长方形。你可以在任何地方开始第一个顶点，最重要的是要以反时针方向来画出顶点。

技巧：你怎样才能知道是否犯了错误，实际上是生成了一个顺时针方向凸面体或者凹面体呢？每个物理引擎的反应都会不一样。有些会通过事先抛出异常告诉你所犯的错误。不过，对于Box2D来说，如果一个移动中的刚体碰到了上述错误的多边形的话，移动中的刚体在快碰到多边形之前就会停下来。如果你在Box2D游戏中看到这样的情况，请检查刚体附近用于碰撞测试的多边形。

使用VertexHelper生成用于定义碰撞多边形的代码

现在我们知道了碰撞多边形的要求，是时候来看一下VertexHelper这个工具了。VertexHelper的源代码是通过GitHub来共享的：

http://github.com/jfahrenkrug/VertexHelper.

点击GitHub上的 Download Source按钮，把下载的文件存在本地电脑上解压缩。然后在Xcode中打开VertexHelper.xcodeproj文件，编译构建然后运行。在运行的VertextHelper的主界面上，它会要求你把精灵图片拖放到牛眼图片的下方。在PhysicsBox2D03项目文件夹中找到 tablebottom.png这张图片，托放到牛眼图片下方。

VertexHelper将会显示拖入的图片，你可以在VertexHelper界面上拖动图片到任何位置。你可以使用Zoom In和Zoom Out按钮对图片进行缩放。你应该在定义碰撞多边形的顶点之前对图片进行移动和缩放，让你可以舒服地进行操作，你至少需要把图片放大两到三倍。

接着，你需要在界面底部设置图片和输出属性，否则VertexHelper将不会允许你添加顶点。Rows/Cols一栏中，在两个输入框中都填入1，因为你没有使用由多个图片平铺起来的图片 - 对于VertexHelper来说，我们的图片只是一个大的瓷砖图片而已。如果你使用的图片是一个纹理贴图集（Texture Atlas），而且纹理贴图集中的图片都是一样大小，并且图片之间的间隔都相同，这种情况下用于编辑同一图片上多个部分的选项才有用处。不过因为你很有可能使用Zwoptex来生成你的纹理贴图集，你很可能需要单独地编辑每个图片的顶点。

将Type设为Box2D，Style设为Initialization，并且把Name改为vertices -这个名字的设定只是我的个人喜好，你可以使用任何你喜欢的名字。然后你就可以点击Edit Mode复选框开始编辑顶点 - 你可以在图片上通过点击来生成新的顶点。

注：请记住：VertexHelper这个工具还不是很完美。一旦你开始添加顶点，你就不能撤销操作，删除或者移动顶点。因此，你的每一次点击都必须准确地落在目标位置上。如果你犯了错误，你将不得不通过 File>New菜单来生成一个新的窗口，然后把图片拖动到界面上重新来一遍。

如果你点击后拖动了鼠标的话，你有可能生成红色的椭圆形形状而不是所需的顶点。这可能是一个Bug或者是隐藏的功能 - 不管怎样，我们的弹球游戏中不需要椭圆形，所以请避免点击后进行拖动。还有一点：请不要点击Sacn按钮。点击Scan按钮会让VertexHelper自动生成一个围绕图片的凸面体（在某些情况下这个功能会很有用处，但是对于我们的弹球游戏来说，自动生成的凸面体派不上用场）。

请记住要以反时针方向创建顶点，并且任何时候都要生成凸面体（Convex）。对于tablebottom.png文件来说，你不能一次性生成一个多边形，因为那样的话你会得到一个凹面体。如**图13-3**所示，你可以通过定义多个多面体达到生成凸面体的目的。

虽然VertexHelper的设计初衷是每个图片中只定义一个多边形，不过你可以一次性定义多个多边形，只要你自己记住定义的多边形和各个多边形的顶点。然后你可以拷贝所有VertexHelper生成的代码，把它们按照不同的多边形分割开 - 只要你知道代码中定义的哪些顶点是为哪些多边形所包含的就可以了。如**图13-3**所示，比如，第一个多边形是最右边的，包含4个顶点。因此你就需要把源代码中与这4个顶点无关的代码删除，并且把 num 这个变量的值从 26 改为 4。接下去的一个多边形包含 6个顶点，你就可以用第 5到第 10行的代码，把 num 变量设为 6。通过这样的方式，你可以为一个图片创建多个多边形，一旦你完成在VertexHelper中的定义工作，你可以通过分割生成的源代码来实现这些碰

撞多边形。只是要注意一点：一定要把相应的 num变量的值修改成一个多边形实际包含的顶点数，否则Box2D可能会崩溃。

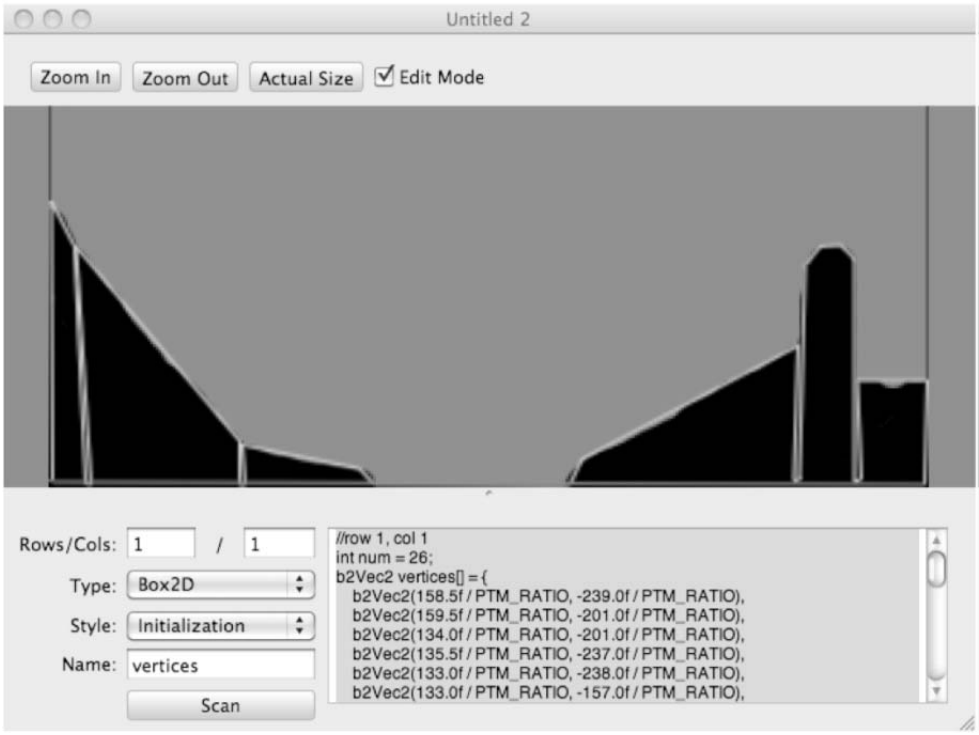

图13-3. 在VertexHelper中编辑碰撞测试用的多边形

现在，所有用于弹球游戏中的碰撞多边形已经创建完成。你可以继续为剩下的图片定义碰撞多边形，它们分别为：flipper-left.png，flipper-right.png，plunger.png和tabletop.png。只有球和减震器使用了Box2D内置的b2CircleShape作为碰撞多边形的定义。

注：你会注意到我设置的顶点的头尾并没有完全叠加起来 - 因为没有必要。只要顶点之间的缝隙和凹槽看上去明显比球本身小，它们对弹球桌面的物理模拟产生的影响可以忽略不计。

生成弹球桌面

在VertexHelper中完成碰撞多边形的定义以后，你需要把VertexHelper生成的代码粘贴到自己的代码中去。在PhysicsBox2D03项目中，我创建了一个名为TableSetup的类，它包含了所有为弹球桌面的静态刚体设置的碰撞多边形。

注：请记住，因为我们使用Box2D作为物理引擎，所以每一个类的实现文件都必须以.mm作为文件类型名称，以避免产生编译错误。

我同时使用了好几个方法为弹球桌面的碰撞多边形设置顶点，但是我不想拿一大堆重复的代码让你无聊，因此我在这里只介绍其中的一个方法，叫作createLanes：

```
-(void) createLanes
{
	//右边的滑道
	{
		// 第一行，第一列
		int num = 5;
		b2Vec2 vertices[] = {
			b2Vec2(100.9f / PTM_RATIO, -143.9f / PTM_RATIO),
			b2Vec2(91.4f / PTM_RATIO, -145.0f / PTM_RATIO),
			b2Vec2(58.2f / PTM_RATIO, -164.4f / PTM_RATIO),
			b2Vec2(76.3f / PTM_RATIO, -185.5f / PTM_RATIO),
			b2Vec2(92.1f / PTM_RATIO, -176.1f / PTM_RATIO),
		};
		[self createStaticBodyWithVertices:vertices numVertices:num];
	}

	// 左边的滑道
	{
		// 第一行，第一列
		int num = 5;
		b2Vec2 vertices[] = {
			b2Vec2(-65.6f / PTM_RATIO, -165.1f / PTM_RATIO),
			b2Vec2(-119.3f / PTM_RATIO, -125.2f / PTM_RATIO),
			b2Vec2(-126.7f / PTM_RATIO, -128.3f / PTM_RATIO),
			b2Vec2(-126.7f / PTM_RATIO, -136.1f / PTM_RATIO),
			b2Vec2(-83.3f / PTM_RATIO, -175.6f / PTM_RATIO)
		};
		[self createStaticBodyWithVertices:vertices numVertices:num];
	}
}
```

上述代码基本上是由VertexHelper生成，然后拷贝粘贴过来的。这些生成的代码包含了所有组成碰撞多边形的顶点，这些顶点信息以Box2D可以处理的格式组织起来。然后这些顶点数组被传给 createStaticBodyWithVertices方法。每个碰撞多边形的定义由一对额外的括号包起来，就在“右边的滑道”和“左边的滑道”下面。每一对括号为变量定义了一个新的范围（scope），这样的话，包含在每一对括号里面的变量在括号外是不能访问的。从if，while和for等的使用中你已经知道了用括号定义变量访问的作用。你也可以不使用任何关键词的情况下使用括号来定义一个新的范围。在我们的例子里，VertexHelper允许你使用相同的变量名，比如上述代码中的 num和 vertices，并且编译器不会报重复定义相同变量的错误。这允许你直接粘贴由VertexHelper生成的代码，而无需做任何更改，也不需要为每个多边形添加一个新的方法。毕竟，你有可能要回去VertexHelper对已经定义过的多边形进行修改，因此能够直接拷贝粘贴生成的代码就显得很重要了。

上述代码中的 createStaticBodyWithVertices方法会使用传入的顶点信息生成静态刚体用于碰撞测试的多边形：

```
-(void) createStaticBodyWithVertices:(b2Vec2[])vertices numVertices:(int)numVertices
{
	b2BodyDef bodyDef;
	bodyDef.position = [Helper toMeters:[Helper screenCenter]];

	b2PolygonShape shape;
	shape.Set(vertices, numVertices);

	b2FixtureDef fixtureDef;
	fixtureDef.shape = &shape;
	fixtureDef.density = 1.0f;
	fixtureDef.friction = 0.2f;
	fixtureDef.restitution = 0.1f;

	b2Body* body = world_->CreateBody(&bodyDef);
	body->CreateFixture(&fixtureDef);
}
```

b2PolygongShape通过调用shape.Set(vertices, numVertices)这个方法，利用传入的顶点信息生成静态刚体的碰撞多边形。我们将b2FixtureDef参数设置成合理的值，作为四周的墙体 - 通常，你不会想把墙体的弹性值设置的很高，因为前提一般都是很硬的，没有多少弹性。

b2BodyDef的位置被设在屏幕的中央。对于弹球桌面，它的背景图片尺寸是320x480像素，然后这张图片被添加到Zwoptex的纹理贴图集里。这样你就可以省下一些贴图用的内存；而且，你总是可以正确地把图片放在屏幕的中央位置。（当然，在iPad上做同样事情的话就需要制作另外的图片了，尺寸为1024x768像素）。我们通过 Helper类中的静态helper方法来获取屏幕中央位置的坐标值。对于 Helper类，我们在之前的章节已经做过讨论。

下面代码中的world_成员变量是用于初始化的临时变量，它存放了 b2World指针，如列表13-1所示：

列表13-1.TableSetup类的头文件

```
#import <Foundation/Foundation.h>
#import "cocos2d.h"
#import "Box2D.h"
@interface TableSetup : CCNode
{
	b2World* world_;
}
+(id) setupTableWithWorld:(b2World*)world;
@end
```

看起来并不太可怕。现在我们来看一下TableSetup类的初始化代码，如**列表13-2**所示：

列表13-2. TableSetup类的初始化

```
-(id) initTableWithWorld:(b2World*)world
{
	if ((self = [super init]))
	{
		// 对world的弱引用
		world_ = world;

		CCSpriteBatchNode* batch = [CCSpriteBatchNode batchNodeWithFile:@"table.png"];
		[self addChild:batch];

		CCSprite* tableTop = [CCSprite spriteWithSpriteFrameName:@"tabletop.png"];
		tableTop.position = [Helper screenCenter];
		[batch addChild:tableTop];

		CCSprite* tableBottom = [CCSprite spriteWithSpriteFrameName:@"tablebottom.png"];
		tableBottom.position = [Helper screenCenter];
		[batch addChild:tableBottom];

		// 生成静态刚体
		[self createTableTopBody];
		[self createTableBottomLeftBody];
		[self createTableBottomRightBody];
		[self createLanes];

		// 初始化完成以后，world就不再需要了
		world_ = NULL;
	}
	return self;
}

+(id) setupTableWithWorld:(b2World*)world
{
	return [[[self alloc] initTableWithWorld:world] autorelease];
}
```

setUpTableWithWorld是个自动释放的静态初始化方法，传入的参数是b2World指针。它会分配内存，然后调用initTableWithWorld方法。这样会把world指针赋值给world_成员变量，但是这只会发生在初始化阶段，而且world_变量只会被保留到初始化完成。使用world_成员变量的只是为了方便而已，这样你就不用把b2World指针传给TableSetup类中的每一个方法了。这个类的作用只是用于设置好弹球桌子。请注意，在结束使用world_成员变量以后，它被设为NULL，而不是nil，因为它是一个C++指针。从技术上来说，NULL和nil是一样的东西，

但是是使用NULL会告诉编译器，这个变量是一个类的指针，没有继承自NSObject。你也可以不用把world_设置成NULL，但是这样做的话你实际上是在告诉编译器你已经使用完world_了，不再需要了。

init方法为桌子的静态背景生成了一个CCSpriteBatchNode，同时生成了tableTop和tableBottom这两个精灵，它们都被放在屏幕中央。接下去的create方法则为弹球桌子设置好静态刚体 - 通过使用在VertexHelper中定义好的碰撞多边形。

TableSetup类是通过PinballTable类来初始化的，这和第12章中Box2D项目里的HelloWorldScene类很相似。它的头文件代码如**列表13-3**所示：

列表13-3. PinballTable的头文件

```
#import "cocos2d.h"
#import "Box2D.h"
#import "GLES-Render.h"
#import "ContactListener.h"
enum
{
	kTagBatchNode,
};

@interface PinballTable : CCLayer
{
	b2World* world;
	ContactListener* contactListener;
	GLESDebugDraw* debugDraw;
}

+(PinballTable*) sharedTable;
// 返回一个仅包含HelloWorld作为子节点的场景
+(id) scene;
-(CCSpriteBatchNode*) getSpriteBatch;
@end
```

上述头文件代码包含了对 ContactListener和GLESDebugDraw类的引用。对于GLESDebugDraw，我将在之后讨论；对于ContactListener，它将在你添加弹球游戏里的活塞（plunger）时发挥作用。现在，让我们来看看列表13-4中的PinballTable类的初始化代码：

列表13-4. 弹球桌面的初始化

```
-(id) init
{
	if ((self = [super init]))
	{
		pinballTableInstance = self;
		[self initBox2dWorld];
```

```
        [self enableBox2dDebugDrawing];

        // 从纹理贴图集中预加载精灵帧
        [[CCSpriteFrameCache sharedSpriteFrameCache]
                                    addSpriteFramesWithFile:@"table.plist"];

        // 将我们的弹球游戏的背景设置为一个明亮的颜色
        CCColorLayer* colorLayer = [CCColorLayer layerWithColor:ccc4(222, 222, 222, 255)];
        [self addChild:colorLayer z:-3];

        // 为所有动态元素设置的批处理精灵节点
        CCSpriteBatchNode* batch =
                [CCSpriteBatchNode batchNodeWithFile:@"table.png" capacity:100];
        [self addChild:batch z:-2 tag:kTagBatchNode];

        // 设置静态元素
        TableSetup* tableSetup = [TableSetup setupTableWithWorld:world];
        [self addChild:tableSetup z:-1];

        [self scheduleUpdate];
    }
    return self;
}
```

Box2D物理引擎的初始化代码已经被移到了单独的方法中：initBox2dWorld。初始化代码和上一章基本上相同，除了一点：我们没有设置屏幕底部的静态刚体，这样动态刚体才能通过底部离开屏幕，掉落到桌子的漏洞中去。

通过加载 tables.plist文件，我使用CCSpriteFrameCache将游戏中使用的纹理贴图集进行初始化。因为桌面上的静态元素时黑色的，所以我使用了一个淡灰色的CCColorLayer来替换默认的黑色背景。之后我也将为动态刚体添加一个CCSpriteBatchNode。通过使用单例设计模式和 pinballTableInstance静态变量，动态刚体就可以访问CCSpriteBatchNode了。

然后，我生成了一个TableSetup类并将其添加到场景中。如果你现在运行这个项目，你会看到一个弹球桌。很好！不过，你怎样才能知道碰撞测试用的多边形已经被正确地摆放好了呢？

Box2D调试渲染

这里我们就要用到 GLESDebugDraw类了。这也是为什么**列表13-4**中的所有子节点的z-order属性都被设置为负的原因。请记住：任何在节点的draw方法中运行的OpenGL ES代码，都会在z-order为0的情况下进行渲染。所以，如果你想把OpenGL ES的输出显示在其它节点之上的话，那些节点的z-order必须为0。

让我们来看一下PinballTable类中的 enableBox2dDebugDrawing方法：

```
-(void) enableBox2dDebugDrawing
{
        debugDraw = new GLESDebugDraw(PTM_RATIO);
        world->SetDebugDraw(debugDraw);

        uint32 flags = b2DebugDraw::e_shapeBit;
        debugDraw->SetFlags(flags);
}
```

首先，通过用PTM_RATIO定义的“像素对米”（pixel-to-meter）的比例，一个GLESDebugDraw类的实例被生成，此实例被保存在PinballTable的成员变量debugDraw中。然后我们通过使用SetDebugDraw方法，将debugDraw这个实例传给Box2D的world。你可以通过在b2DebugDraw中添加bits的方式来定义你想画的东西，其中的e_shapeBit是最重要的，因为它会把所有刚体的碰撞多边形都渲染显示出来。

光那样还不够 — 你必须把PinballTable类的draw方法覆盖调，然后调用debugDraw->DrawDebugData()方法来实际绘制调试信息。因为你肯定不想让用户看到这些调试信息，所以draw方法是包含在＃ifdef DEBUG...＃endif语句之间：

```
#ifdef DEBUG
-(void) draw
{
        glDisable(GL_TEXTURE_2D);
        glDisableClientState(GL_COLOR_ARRAY);
        glDisableClientState(GL_TEXTURE_COORD_ARRAY);

        world->DrawDebugData();

        // 重置默认的GL状态
        glEnable(GL_TEXTURE_2D);
        glEnableClientState(GL_COLOR_ARRAY);
        glEnableClientState(GL_TEXTURE_COORD_ARRAY);
}
#endif
```

为了正确渲染调试信息，OpenGL ES的一些状态需要被禁用。不过你不需要关心这件事情，不过如果你感兴趣想学习更多相关的知识，你可以参考OpenGL ES 1.1的参考手册：

www.khronos.org/opengles/sdk/1.1/docs/man.

添加动态元素

到目前为止，弹球桌上还没有会动的元素，所以没有会弹来弹去的东西。不过，在我添加球和碰撞杆之前（还有翻板与活塞），我要先介绍一下BodyNode类，这样当你用cocos2d精灵去配合动态刚体时会觉得舒服一点。

BodyNode类

BodyNode类的理念是：用一个自给自足的对象为所有动态刚体类服务。到目前为止，我只是把CCSprite添加到了刚体的userData域中。但是如果你想和由精灵代表的类互动的话 — 比如，在ContactListener方法里。你就不能和刚体对象互动了，因为你有的只是一个CCSprite对象而已。

为了解决这个问题，我在PhysicsBox2D04项目中创建了BodyNode类。BodyNode继承自CCNode，刚体和精灵都是它的成员变量。对刚体的引用是为了让子节点可以方便地访问它。BodyNode类也包含了一个CCSprite指针，这样的话BodyNode类也可以在PinballTable类的update方法中把它自己显示在屏幕上。当所有动态刚体都继承自BodyNode类以后，你将会拥有一个统一的类来调用动态刚体。你也可以进一步地探索它的类型 — 比如，使用所有继承自NSObject的类都支持的 isKindOfClass方法。

另外，BodyNode类的头文件中也包含了一般都需要用到的头文件，比如Box2d.h和Helper.h（如**列表13-5**所示）：

列表13-5. BodyNode的头文件

```
#import <Foundation/Foundation.h>
#import "cocos2d.h"
#import "Box2D.h"
#import "Helper.h"
#import "Constants.h"
#import "PinballTable.h"

@interface BodyNode : CCNode
{
        b2Body* body;
        CCSprite* sprite;
}
@property (readonly, nonatomic) b2Body* body;
@property (readonly, nonatomic) CCSprite* sprite;

-(void) createBodyInWorld:(b2World*)world bodyDef:(b2BodyDef*)bodyDef
fixtureDef:(b2FixtureDef*)fixtureDef spriteFrameName:(NSString*)spriteFrameName;

-(void) removeSprite;
-(void) removeBody;
@end
```

BodyNode也为它的两个成员变量，body和sprite，提供了访问方法，这样它们就可以被其它类方便地访问到了。createBodyInWorld方法是继承自BodyNode的类用来初始化刚体时使用的。此方法的参数包括b2BodyDef，b2FixtureDef和通过使用来自纹理贴图集的精灵帧名称所创建的CCSprite。列表13-6展示了实现代码：

列表13-6. BodyNode类的实现代码

```
#import "BodyNode.h"
@implementation BodyNode

@synthesize body;
@synthesize sprite;

-(void) createBodyInWorld:(b2World*)world bodyDef:(b2BodyDef*)bodyDef
                                                fixtureDef:(b2FixtureDef*)fixtureDef
                                                spriteFrameName:(NSString*)spriteFrameName
{
        NSAssert(world != NULL, @"world is null!");
        NSAssert(bodyDef != NULL, @"bodyDef is null!");
        NSAssert(spriteFrameName != nil, @"spriteFrameName is nil!");

        [self removeSprite];
        [self removeBody];

        CCSpriteBatchNode* batch = [[PinballTable sharedTable] getSpriteBatch];
        sprite = [CCSprite spriteWithSpriteFrameName:spriteFrameName];
        [batch addChild:sprite];

        body = world->CreateBody(bodyDef);
        body->SetUserData(self);

        if (fixtureDef != NULL)
        {
                body->CreateFixture(fixtureDef);
        }
}

-(void) removeSprite
{
        CCSpriteBatchNode* batch = [[PinballTable sharedTable] getSpriteBatch];
        if (sprite != nil && [batch.children containsObject:sprite])
        {
                [batch.children removeObject:sprite];
                sprite = nil;
        }
```

```
}

-(void) removeBody
{
	if (body != NULL)
	{
		body->GetWorld()->DestroyBody(body);
		body = NULL;
	}
}

-(void) dealloc
{
	[self removeSprite];
	[self removeBody];
	[super dealloc];
}
@end
```

第一眼看上去，BodyNode只是在恰当的时候简单地调用了world的CreateBody方法和刚体的CreateFixture方法。刚体被储存在body成员变量中。不过，这个方法的重点之处在于：BodyNode为你管理着精灵和刚体的内存分配和释放，它利用同一个纹理贴图集生成精灵，然后把精灵放入一个通用的批量处理精灵中。和之前一样，我们通过body->SetUserData(self)把当前的BodyNode实例作为user data指针赋值给刚体。

如果系统不再需要某个继承自BodyNode的类时 — 例如，如果你将这个不再需要的子节点从cocos2d的父节点移除的话，BodyNode会自动帮你移除相应的Box2D刚体，也会帮你把所用到的精灵从CCSpriteBatchNode中移除。这个特点也让createBodyInWorld方法可以被重复使用。因为如果你想改变某个已经存在的刚体，你只要用带不同值的参数再次调用这个方法，它就会为你移除旧的刚体和精灵，生成新的刚体和精灵。

继承自BodyNode的类现在只要正确地设置b2BodyDef和b2FixtureDef这两个域，并且提供正确的精灵帧名称就可以了。

位于PinballTable类中的update方法也已经被重写，因为user data现在接受的是BodyNode而不是CCSprite了：

```
-(void) update:(ccTime)delta
{
	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())
        {
                BodyNode* bodyNode = (BodyNode*)body->GetUserData();
                if (bodyNode != NULL && bodyNode.sprite != nil)
                {
                        bodyNode.sprite.position = [Helper toPixels:body->GetPosition()];
                        float angle = body->GetAngle();
                        bodyNode.sprite.rotation = -(CC_RADIANS_TO_DEGREES(angle));
                }
        }
}
```

现在user data指针是一个BodyNode对象而不再是CCSprite。我们现在通过bodyNode.sprite属性来访问精灵。

技巧：CCNode类也有一个userData属性，你可以像使用b2Body的userData域一样来使用它。

弹球

你能想像一个没有弹球的弹球游戏吗？我不能。所以让我们来添加一个弹球，看看它的实现方法。Ball类继承自BodyNode，而且也实现了CCTargetedTouchDelegate协议（处于试验的目的）（如**列表13-7**所示）：

列表13-7. Ball类的头文件

```
#import "BodyNode.h"
@interface Ball : BodyNode <CCTargetedTouchDelegate>
{
        bool moveToFinger;
        CGPoint fingerLocation;
}
+(id) ballWithWorld:(b2World*)world;
@end
```

这里面有通常的静态初始化方法 ballWithWorld，它接受一个b2World指针作为参数。接下去是成员变量 moveToFinger，它被用于弹球是否应该向用户触摸的位置移动。CGPoint类型的成员变量 fingerLocation用于指定用户手指触摸到的实际位置。只要我们的弹球游戏没有包含其它互动的元素，我们可以使用上述成员变量尝试一些有趣的事情。**列表13-8**展示了弹球的初始化和内存释放方法：

列表13-8. Ball类的init和dealloc方法

```
-(id) initWithWorld:(b2World*)world
{
```

```
        if ((self = [super init]))
        {
                [self createBallInWorld:world];
                [[CCTouchDispatcher sharedDispatcher] addTargetedDelegate:self
                                                                priority:0 swallowsTouches:NO];
                [self scheduleUpdate];
        }
        return self;
}

+(id) ballWithWorld:(b2World*)world
{
        return [[[self alloc] initWithWorld:world] autorelease];
}

-(void) dealloc
{
        [[CCTouchDispatcher sharedDispatcher] removeDelegate:self];
        [super dealloc];
}
```

生成弹球

上述初始化和内存释放方法很容易理解，所以我们直接开始讨论createBallInWorld方法（如**列表13-9**所示），它会将弹球初始化，然后为弹球设置所有参数。它也会把弹球放在活塞位置的上方：

列表13-9. 创建弹球

```
-(void) createBallInWorld:(b2World*)world
{
        CGSize screenSize = [[CCDirector sharedDirector] winSize];
        float randomOffset = CCRANDOM_0_1() * 10.0f - 5.0f;
        CGPoint startPos = CGPointMake(screenSize.width - 15 + randomOffset, 80);

        b2BodyDef bodyDef;
        bodyDef.type = b2_dynamicBody;
        bodyDef.position = [Helper toMeters:startPos];
        bodyDef.angularDamping = 0.9f;

        NSString* spriteFrameName = @"ball.png";
        CCSprite* tempSprite = [CCSprite spriteWithSpriteFrameName:spriteFrameName];

        b2CircleShape shape;
        float radiusInMeters = (tempSprite.contentSize.width / PTM_RATIO) * 0.5f;
        shape.m_radius = radiusInMeters;
```

```
        b2FixtureDef fixtureDef;
        fixtureDef.shape = &shape;
        fixtureDef.density = 0.8f;
        fixtureDef.friction = 0.7f;
        fixtureDef.restitution = 0.3f;

        [super createBodyInWorld:world bodyDef:&bodyDef fixtureDef:&fixtureDef
                                                    spriteFrameName:spriteFrameName];

        sprite.color = ccRED;
}
```

让我来指出这个方法中到目前为止我还没有讨论过的令人好奇的地方。我将b2BodyDef的一个名为angularDamping的域设为0.9f，它会让球对转弯的动作产生更大的阻力。这样的话，我们的弹球在滑过弹球桌面的时候不会产生很多的自身旋转，这对于用金属制作的有一定重量的弹球来说是标准的移动方式。

因为b2CircleShape的半径取决于图片的大小，我创建了一个临时的精灵。这个精灵的图片宽度被除以“像素对米”（pixel-to-meter）的比例，然后乘以0.5f，因为我们是通过圆圈的半径而不是直径来定义弹球的。

b2FixtureDef使用了我认为合适的密度，摩擦力和弹力的值来代表弹球的各个属性。在这里我没有花太多时间来微调它们。微调物理引擎的值通常很费时间和精力，并且要求你仔细考虑每次修改可能会带来的影响。不过设计师和程序员经常对上述情况估计不足。对于我们的弹球游戏，虽然我们可以比较好的模拟游戏中的物理效果，但是要达到这些效果之前花费和挺多功夫做微调的。

最终，我们传入了bodyDef，fixtureDef和spriteFrameName给位于BodyNode类中的超级（super）方法createBodyInWorld，生成了刚体和精灵。在调用createBodyInWorld方法以后，你就可以使用body和sprite这两个成员变量了，因为它们现在都已经被初始化成功了。例如，我可以将ball的颜色设为红色了。

实际上，每次弹球掉落到桌子的凹槽离开桌子以后，createBallInWorld方法就会被调用，系统将会把它的位置设为初始位置。你也可以使用以下代码直接设置刚体的位置：

```
body->SetTransform(newPosition);
```

不过，上述简短代码不好的地方是：它只能改变刚体的位置，但是不会重设刚体的当前角度和速度。因此，最好的方式就是再次调用createBodyInWorld方法，它会在生成新的刚体和精灵之前，帮你从world里移除任何已经存在的刚体，还有任何已经存在于批处理精灵节点中的精灵。

要让弹球出现在桌面上，你必须将其添加到场景中。这可以在TableSetup类中的init方法里实现，代码如下：

```
Ball* ball = [Ball ballWithWorld:world];
[self addChild:ball z:-1];
```

让弹球移动

到目前为止，我们的弹球除了会往下掉落外，其它什么都不会。我们需要一个控制弹球的方式。弹球的类里面实现了CCTargetedTouchDelegate，并且已经把自己注册为可以接收触摸事件。让我们来看一下触摸代理方法都能做些什么：

```
-(BOOL) ccTouchBegan:(UITouch *)touch withEvent:(UIEvent *)event
{
	moveToFinger = YES;
	fingerLocation = [Helper locationFromTouch:touch];
	return YES;
}

-(void) ccTouchMoved:(UITouch *)touch withEvent:(UIEvent *)event
{
	fingerLocation = [Helper locationFromTouch:touch];
}

-(void) ccTouchEnded:(UITouch *)touch withEvent:(UIEvent *)event
{
	moveToFinger = NO;
}
```

这些方法规定了当手指触摸到屏幕上某处时，弹球会朝着手指触摸的位置移动；当手指在屏幕移动时，fingerLocation的值则一直会被更新。

下面，让我们很快地看一下update方法：

```
-(void) update:(ccTime)delta
{
	if (moveToFinger == YES)
	{
		[self applyForceTowardsFinger];
	}

	if (sprite.position.y < -(sprite.contentSize.height * 10))
	{
		// 生成一个新的弹球，并且把现有的弹球移除
		[self createBallInWorld:body->GetWorld()];
	}
}
```

我将很快开始讨论applyForceTowardsFinger方法。不过现在，请注意在上述代

码中我在生成新球之前，测试了弹球是否已经掉落到凹槽之中。我用精灵的y轴坐标值与精灵图片的高度乘以10做比较。为什么要做个乘法呢？那只是为了给人球要掉落一段距离才能到达凹槽底部的感觉。当球掉落足够的距离以后，我们再次调用createBallInWorld方法来生成新球和开始新一轮的游戏。

现在我们来看一下 applyForceTowardsFinger方法，它会让球加速度向手指移动，如**列表13-10**所示：

列表13-10. 让球向手指触摸的位置做加速度移动

```
-(void) applyForceTowardsFinger
{
        b2Vec2 bodyPos = body->GetWorldCenter();
        b2Vec2 fingerPos = [Helper toMeters:fingerLocation];

        b2Vec2 bodyToFinger = fingerPos - bodyPos;
        bodyToFinger.Normalize();

        b2Vec2 force = 2.0f * bodyToFinger;
        body->ApplyForce(force, body->GetWorldCenter());
}
```

我们首先得到刚体位置和手指触摸处的位置，然后用手指位置减去刚体位置以得到它们之间的距离。b2Vec2这个结构使用了名为“运算符重载”（Operator Overloading）的技术，这种技术可以允许你把两个或者多个b2Vec2结构进行减法，加法和乘法操作。运算符重载是一种C++的语言功能；Objective-C中没有这种功能 — 所以你不能够把CGPoint变量做减法，加法和乘法操作。

矢量bodyToFinger是从刚体的位置指向手指的触摸位置的。当我们在bodyToFinger上调用Normalize函数时，bodyToFinger就被转换成了unit vector。unit vector就是长度为1的矢量 — 或者是1个单位(unit)。转换成unit vector以后，你就可以将它与固定的系数相乘，在上述代码中，我们将其乘以2.0f，把它长度增加一倍。然后你就可以把这个值应用到ApplyForce方法中，作为应用到刚体中心的外部力量。你也可以不使用刚体的中心位置，而是用别的位置信息；不过，那样的话刚体就会开始自我旋转了。

上述代码得到的结果是球会向你的手指在屏幕上的触摸位置加速移动。通常，球会移动的超过目标位置，慢下来，然后返回。有点像太阳施加在我们星球上的重力牵引力一样。不过，对于我这样对天文学感兴趣的人来说，我必须纠正我自己的说法。重力是一种以通过重力相互牵引的两个物体之间的距离的平方进行衰减的力量。因此，如果你想在你的游戏中模拟更加真实的重力效果的话，你可以用**列表13-11**中的代码来替换applyForceTowardsFinger中的代码：

列表13-11. 模拟重力牵引的效果

```
b2Vec2 bodyPos = body->GetWorldCenter();
b2Vec2 fingerPos = [Helper toMeters:fingerLocation];
float distance = bodyToFinger.Normalize();
```

```
//“真实”的重力是以相互作用的两个物体之间距离的平方来进行衰减的
float distanceSquared = distance * distance;
b2Vec2 force = ((1.0f / distanceSquared) * 20.0f) * bodyToFinger;
body->ApplyForce(force, body->GetWorldCenter());
```

上述代码中的20.0f是一个拥有魔力的数字。它可以让重力的拉动作用变得明显。现在，如果弹球离开手指触摸位置的距离短的话，球的速度会很快；而如果手指触摸的位置离开弹球相对较远的话，弹球几乎不会移动。

虽然applyForceTowardsFinger代码只是作为临时的控制方式，但是你可以用**列表13-11**中的重力代码创建弹球桌上的磁性物体。

碰撞体（The Bumpers）

现在，你已经有了一个会随着你手指移动而移动的弹球，让我们把游戏弄的更有趣一些。我将在这里介绍游戏中的碰撞体。什么是碰撞体呢？它们是那种圆形的，蘑菇状的物体。在弹球碰到它们时会被弹开。

列表13-12 展示了Bumper类的头文件代码：

列表13-12. Bumper类的头文件

```
#import "BodyNode.h"
@interface Bumper : BodyNode
{
}

+(id) bumperWithWorld:(b2World*)world position:(CGPoint)pos;
@end
```

当然，Bumper类继承自BodyNode。它的初始化代码和**列表13-9**中初始化弹球的代码很相像，因此我只讲在**列表13-13**中专注于重要的部分：

列表13-13. 使用高弹性力设置初始化Bumper类

```
-(id) initWithWorld:(b2World*)world position:(CGPoint)pos
{
        if ((self = [super init]))
        {
                ...
                b2FixtureDef fixtureDef;
                fixtureDef.shape = &circleShape;
                fixtureDef.density = 1.0f;
                fixtureDef.friction = 0.8f;

                // restitution > 1 可以让球弹出的速度比碰撞前的速度更快
                fixtureDef.restitution = 1.5f;
                [super createBodyInWorld:world bodyDef:&bodyDef fixtureDef:&fixtureDef
                                                        spriteFrameName:spriteFrameName];
                sprite.color = ccORANGE;
```

```
        }
        return self;
}
```

Bumper类的唯一一个主要的组成部分是将restitution参数设置为大于1.0f —在我们的例子中设为1.5f。这个设置会让任何撞上碰撞体的刚体所得到的反弹力比它对碰撞体的撞击力要大50%。所得到的结果是在现实物理世界中不可能发生的：刚体撞击物体以后弹回的速度比它撞击前还要快。当然，在我们的例子力，Box2D已经为我们设置好了所有的逻辑。

剩下需要做的就是把以下代码放到TableSetup类的init方法中，以完成碰撞体的设置。你可以随意安排这些碰撞体的位置：

```
// 添加一些碰撞体
[self addBumperAt:CGPointMake(150, 330)];
[self addBumperAt:CGPointMake(100, 390)];
[self addBumperAt:CGPointMake(230, 380)];
[self addBumperAt:CGPointMake(40, 350)];
[self addBumperAt:CGPointMake(280, 300)];
[self addBumperAt:CGPointMake(70, 280)];
[self addBumperAt:CGPointMake(240, 250)];
[self addBumperAt:CGPointMake(170, 280)];
[self addBumperAt:CGPointMake(160, 400)];
[self addBumperAt:CGPointMake(15, 160)];
```

试着运行一下PhysicsBox2D03项目，感受一下这些碰撞体的作用。

活塞（The Plunger）

我不想把你对弹球的控制移除，但是我必须那么做。我们现在需要添加活塞了，所以如果你还能够用手指控制弹球的话，那我们的活塞就不能很好地工作了。因此我们要到Ball类的update方法中，把调用applyForceTowardsFinger的那几行代码给注释掉：

```
if (moveToFinger == YES)
{
        // 目前不需要
        // [self applyForceTowardsFinger];
}
```

现在你可以添加Plunger类了，我已经在PhysicsBox2D05项目中添加了Plunger类。**列表13-14**展示了Plunger类的头文件，继承自BodyNode：

列表13-14. Plunger类的头文件

```
#import "BodyNode.h"
@interface Plunger : BodyNode
{
        b2PrismaticJoint* joint;
```

```
        bool doPlunge;
        ccTime plungeTime;
}
@property (nonatomic) bool doPlunge;
+(id) plungerWithWorld:(b2World*)world;
@end
```

上述代码中有一个名为b2PrismaticJoint的成员变量，它会让自己往上移动。柱状关节（Prismatic Joint）只允许你朝一个方向移动 — 单筒望远镜是一个应用柱状关节的很好的例子。你只能朝一个方向移动位于大管子中的小管子来调整望远镜。

如列表13-15所示，活塞的初始化也很直接。活塞的位置需要一点微调，同时我把它的摩擦力和密度设置为极限值，这样弹球就不会在碰到活塞的时候弹跳出去了，这可以保证平滑的发射动作。活塞的形状通过VertexHelper创建。

列表13-15. 初始化活塞

```
-(id) initWithWorld:(b2World*)world
{
        if ((self = [super init]))
        {
                CGSize screenSize = [[CCDirector sharedDirector] winSize];
                CGPoint plungerPos = CGPointMake(screenSize.width - 13, 32);

                b2BodyDef bodyDef;
                bodyDef.type = b2_dynamicBody;
                bodyDef.position = [Helper toMeters:plungerPos];

                b2PolygonShape shape;
                int num = 4;
                b2Vec2 vertices[] = {
                        b2Vec2(10.5f / PTM_RATIO, 10.6f / PTM_RATIO),
                        b2Vec2(11.8f / PTM_RATIO, 18.1f / PTM_RATIO),
                        b2Vec2(-11.9f / PTM_RATIO, 18.3f / PTM_RATIO),
                        b2Vec2(-10.5f / PTM_RATIO, 10.8f / PTM_RATIO)
                };
                shape.Set(vertices, num);

                b2FixtureDef fixtureDef;
                fixtureDef.shape = &shape;
                fixtureDef.density = 1.0f;
                fixtureDef.friction = 0.99f;
                fixtureDef.restitution = 0.01f;
```

```
            [super createBodyInWorld:world bodyDef:&bodyDef fixtureDef:&fixtureDef
                                                    spriteFrameName:@"plunger.png"];
            sprite.position = plungerPos;
            [self attachPlunger];
            [self scheduleUpdate];
        }
        return self;
}
```

更有意思的是对attachPlunger方法的调用和在此方法中创建柱状关节的代码，如**列表13-16**所示：

列表13-16. 生成Plunger的柱状关节

```
-(void) attachPlunger
{
        // 生成一个隐形的静态刚体作为固定关节之用
        b2BodyDef bodyDef;
        bodyDef.position = body->GetWorldCenter();
        b2Body* staticBody = body->GetWorld()->CreateBody(&bodyDef);

        // 生成一个柱状关节让活塞可以做上下运动
        b2PrismaticJointDef jointDef;
        b2Vec2 worldAxis(0.0f, 1.0f);
        jointDef.Initialize(staticBody, body, body->GetWorldCenter(), worldAxis);
        jointDef.lowerTranslation = 0.0f;
        jointDef.upperTranslation = 0.75f;
        jointDef.enableLimit = true;
        jointDef.maxMotorForce = 60.0f;
        jointDef.motorSpeed = 20.0f;
        jointDef.enableMotor = false;

        joint = (b2PrismaticJoint*)body->GetWorld()->CreateJoint(&jointDef);
}
```

首先，在活塞动态刚体的位置生成一个静态刚体。这个静态刚体（staticBody）将会固定住活塞。

worldAixs会把柱状关节的移动限制在Y轴方向（比如，上下移动）。这里，worldAxis用一个值为0.0f和1.0f的普通矢量作为表示；当Y轴被设置为1.0f时，worldAxis就与Y轴平行。如果你把X和Y轴都设为0.5f的话，worldAxis就是45度角。b2PrismaticJointDef由staticBody和活塞的动态刚体的中心位置来进行初始化。worldAixs被用作关节的定位点（anchor point），这样就把柱状关节的运动方向限制在Y轴方向了。

接下去就是对一组参数的设置。lowerTranslation和upperTranslation决定着

活塞在Y轴上的上下移动距离。在我们的例子力，活塞可以向上移动0.75f米，也就是刚好24像素。enableLimit域被设置为true，这样的话连接这活塞的刚体的移动也受到相同的限制。因为静态刚体不会移动，所以活塞刚体移动时将会移动到设置的极限值。如果活塞刚体和关节刚体都是动态刚体的话，两个刚体都将会移动，那就不是我们想要达到的效果了。

接着，我把maxMotorForce的值设为60（这里所用的单位是“牛顿-米”(Newton-meters)，它是用于衡量扭矩的单位）。在Chipmunk中这被成为body's moment。maxMotorForce的值限制了关节移动的扭矩或能量。motorSpeed的值则决定这在多长时间内maxMotorForce的值会到达顶峰。这两个值是通过试验才得到的。现在，弹球会以大概正确的速度被弹射出去了。enableMotor一开始被设置为false，因为我只想在有弹球接触到活塞时，才让活塞进行弹射。

然后，我们使用world的CreateJoint方法生成柱状关节，并将它保存在joint成员变量中。因为CreateJoint方法返回的时一个b2Joint指针，所以这个指针在赋值之前必须被转换为b2PrismaticJoint指针。

请注意，我们并不需要把由Plunger类保存为成员变量的joint对象移除。joint对象会在任何一个连接着它的刚体被移除时自动销毁，也就是BodyNode的dealloc方法销毁刚体时joint会被销毁。

当弹球碰到活塞时进行发射

弹球的发射实际上时自动的。在BeginContact方法中，**列表13-17**中的代码把活塞的doPlunge属性设置为YES，这个属性会被用于测试何时让活塞向上移动：

列表13-17.决定何时发射活塞

```
void ContactListener::BeginContact(b2Contact* contact)
{
	b2Body* bodyA = contact->GetFixtureA()->GetBody();
	b2Body* bodyB = contact->GetFixtureB()->GetBody();
	BodyNode* bodyNodeA = (BodyNode*)bodyA->GetUserData();
	BodyNode* bodyNodeB = (BodyNode*)bodyB->GetUserData();

	if ([bodyNodeA isKindOfClass:[Plunger class]] && [bodyNodeB isKindOfClass:[Ball class]])
	{
		Plunger* plunger = (Plunger*)bodyNodeA;
		plunger.doPlunge = YES;
	}
	else if ([bodyNodeB isKindOfClass:[Plunger class]] && [bodyNodeA isKindOfClass:[Ball class]])
	{
		Plunger* plunger = (Plunger*)bodyNodeB;
		plunger.doPlunge = YES;
	}
```

```
}
```

因为活塞和球都有可能时body A或者B，所以我们必须检查两种情况。只有在弹球碰到活塞时，我们才想把活塞的doPlunge属性设置为YES。因为我们把BodyNode实例保存在刚体的userData域中，所以我们只要通过使用isKindOfClass方法来检查BodyNode类就行了。

那么当doPlunge被设置为YES后又会发生什么事情呢？活塞的update方法会一直检查doPlunge是否为YES，如果时YES，它会把对doPlunge设为NO，然后启动关节的马达（motor）。另一个预约好的方法会把关节的马达关闭，这样重力可以参与进来，让活塞在一小段时间后开始做向下的运动：

```
-(void) update:(ccTime)delta
{
	if (doPlunge == YES)
	{
		doPlunge = NO;
		joint->EnableMotor(YES);

		// 预约停止马达的方法
		[self unschedule:_cmd];
		[self schedule:@selector(endPlunge:) interval:0.5f];
	}
}
```

endPlunge方法只是简单地把原先的方法预约解除，然后把关节的马达关闭。

```
-(void) endPlunge:(ccTime)delta
{
	[self unschedule:_cmd];
	joint->EnableMotor(NO);
}
```

和其它桌子上的元素一样，我们生成活塞并把它添加到TableSetup类的init方法中：

```
Plunger* plunger = [Plunger plungerWithWorld:world];
[self addChild:plunger z:-2];
```

翻板(The Flippers)

最后一个元素时翻板，你可以用它们来控制弹球的动作。两个翻板是通过点击屏幕的左侧或着右侧来控制的，如**列表13-18**所示：

列表13-18. Flipper的头文件

```
#import "BodyNode.h"
typedef enum
{
	FlipperLeft,
```

```
        FlipperRight,
} EFlipperType;

@interface Flipper : BodyNode <CCTargetedTouchDelegate>
{
        EFlipperType type;
        b2RevoluteJoint* joint;
        float totalTime;
}
+(id) flipperWithWorld:(b2World*)world flipperType:(EFlipperType)flipperType;
@end
```

每个翻板使用b2RevoluteJoint来定位的。请看一下Flipper的initWithWorld方法，了解翻板是如何被生成的，如**列表13-19**所示：

列表13-19. 生成Flipper

```
-(id) initWithWorld:(b2World*)world flipperType:(EFlipperType)flipperType
{
        if ((self = [super init]))
        {
                type = flipperType;

                CGSize screenSize = [[CCDirector sharedDirector] winSize];
                CGPoint flipperPos = CGPointMake(screenSize.width / 2 - 48, 55);
                if (type == FlipperRight)
                {
                        flipperPos = CGPointMake(screenSize.width / 2 + 40, 55);
                }

                // 生成刚体的定义，它是一个动态刚体
                b2BodyDef bodyDef;
                bodyDef.type = b2_dynamicBody;
                bodyDef.position = [Helper toMeters:flipperPos];

                // 定义动态刚体的fixture
                b2FixtureDef fixtureDef;
                fixtureDef.density = 1.0f;
                fixtureDef.friction = 0.99f;
                fixtureDef.restitution = 0.1f;

                b2PolygonShape shape;
                b2Vec2 revolutePoint;
                b2Vec2 revolutePointOffset = b2Vec2(0.5f, 0.0f);

                if (type == FlipperLeft)
                {
```

```
                int numVertices = 4;
                b2Vec2 vertices[] = {
                        b2Vec2(-20.5f / PTM_RATIO, -1.7f / PTM_RATIO),
                        b2Vec2(25.0f / PTM_RATIO, -25.5f / PTM_RATIO),
                        b2Vec2(29.5f / PTM_RATIO, -23.7f / PTM_RATIO),
                        b2Vec2(-10.2f / PTM_RATIO, 12.5f / PTM_RATIO)
                };
                shape.Set(vertices, numVertices);
                revolutePoint = bodyDef.position - revolutePointOffset;
        }
        else
        {
                int numVertices = 4;
                b2Vec2 vertices[] = {
                        b2Vec2(11.0f / PTM_RATIO, 12.5f / PTM_RATIO),
                        b2Vec2(-29.5f / PTM_RATIO, -23.5f / PTM_RATIO),
                        b2Vec2(-23.2f / PTM_RATIO, -25.5f / PTM_RATIO),
                        b2Vec2(19.7f / PTM_RATIO, -1.7f / PTM_RATIO)
                };
                shape.Set(vertices, numVertices);
                revolutePoint = bodyDef.position + revolutePointOffset;
        }
        fixtureDef.shape = &shape;
        [super createBodyInWorld:world bodyDef:&bodyDef fixtureDef:&fixtureDef
                                                        spriteFrameName:@"flipper-left.png"];

        if (type == FlipperRight)
        {
                sprite.flipX = YES;
        }

        [self attachFlipperAt:revolutePoint];
        [[CCTouchDispatcher sharedDispatcher] addTargetedDelegate:self
                                                                priority:0 swallowsTouches:NO];
    }
    return self;
}
```

我用左边翻板的代码作为参考，这样的话，如果代码中flipperType被设置为FlipperRight的话，我只要把右边翻板的位置改变，然后把精灵翻转过来就可以了。同样的方法被应用在attachFlipperAt方法中。如**列表13-20**所示，此方法会生成用于翻板的关节。为了改变翻板的转向和旋转的上限值，我使用了一个名为revolutePoint的变量，用它来决定各个翻板的旋转点。因为翻板是不应该围绕它自己的中心点进行旋转的：

列表13-20. 生成翻板的旋转关节（Revolute Joint）

```
-(void) attachFlipperAt:(b2Vec2)pos
{
	// 生成一个隐形的静态刚体，将翻板固定到上面
	b2BodyDef bodyDef;
	bodyDef.position = pos;
	b2Body* staticBody = body->GetWorld()->CreateBody(&bodyDef);

	b2RevoluteJointDef jointDef;
	jointDef.Initialize(staticBody, body, staticBody->GetWorldCenter());
	jointDef.lowerAngle = 0.0f;
	jointDef.upperAngle = CC_DEGREES_TO_RADIANS(70);
	jointDef.enableLimit = true;
	jointDef.maxMotorTorque = 30.0f;
	jointDef.motorSpeed = -20.0f;
	jointDef.enableMotor = true;

	if (type == FlipperRight)
	{
		jointDef.motorSpeed *= -1;
		jointDef.lowerAngle = -jointDef.upperAngle;
		jointDef.upperAngle = 0.0f;
	}
	joint = (b2RevoluteJoint*)body->GetWorld()->CreateJoint(&jointDef);
}
```

生成静态刚体的目的是把翻板固定到上面，以使之无法移动。
b2RevoluteJointDef使用lowerAngle和upperAngle作为翻板的旋转限制（以弧度为单位）。在这里我把upperAngle的值设为70度，并且使用了cocos2d提供的宏命令CC_DEGREES_TO_RADIANS把度转换成为弧度。

旋转关节（Revolute Joint）拥有maxMotorTorque和motorSpeed这两个域，它们被用来定义翻板的旋转速度和翻板对旋转动作的灵敏度。不过，在我们的例子中，马达是一直处于开启状态的。当翻板处于闲置状态时，马达会迫使翻板处于下方不动，这样的话就不会碰到掉落的弹球了。

在ccTouchBegan方法中，我们获取了手指触摸的位置信息，然后在反转马达之前用 isTouchForMe方法进行验证：

```
-(BOOL) ccTouchBegan:(UITouch *)touch withEvent:(UIEvent *)event
{
	BOOL touchHandled = NO;
	CGPoint location = [Helper locationFromTouch:touch];
	if ([self isTouchForMe:location])
```

```
        {
                touchHandled = YES;
                [self reverseMotor];
        }
        return touchHandled;
}

-(void) ccTouchEnded:(UITouch *)touch withEvent:(UIEvent *)event
{
        CGPoint location = [Helper locationFromTouch:touch];
        if ([self isTouchForMe:location])
        {
                [self reverseMotor];
        }
}
```

isTouchForMe方法用于检查哪一侧的屏幕接收到了触摸事件，并且检查当前的翻板实例是否可以正确地响应触摸事件：

```
-(bool) isTouchForMe:(CGPoint)location
{
        if (type == FlipperLeft && location.x < [Helper screenCenter].x)
        {
                return YES;
        }
        else if (type == FlipperRight && location.x > [Helper screenCenter].x)
        {
                return YES;
        }
        return NO;
}
```

然后，我们通过反转马达让翻板弹起。当触摸事件结束时让翻板弹回原位：

```
-(void) reverseMotor
{
        joint->SetMotorSpeed(joint->GetMotorSpeed() * -1);
}
```

剩下的就只是物理模拟了。如果弹球碰到了翻板，并且你触摸了正确的一侧，翻板会加速向上，将球推出去。取决于球在翻板上的碰撞位置，球在上弹的时候会有不同的角度。

结语

本章我们学习了如何使用VertexHelper这个工具为弹球游戏中的刚体定义碰撞测试用的多边形。完成球的设置以后，我展示和如何模拟出让球向手指触摸的地方加速移动，包括如何模拟重力或磁力的效果。

我希望学习完本章后，你会认识到物理模拟所带来的乐趣（不管你在真实的物理课上经历了什么困难）。不过，你并没有在物理课上创造出一款弹球游戏，对吗？

如果你想学习更多的知识 — 例如，使用更多种类的关节或着更多地控制碰撞过程 — 你可以参考Box2D的手册：

www.box2d.org/manual.html

另外，euguo你想了解更多关于各个类和结构的信息，你应该参考Box2D API参考。API参考没有被放在网上，但是你可以在Box2D源码包中的Documentation文件夹里找到。你可以在以下网址下载Box2D源码包：

http://code.google.com/p/box2d

如果你需要Box2D方面的帮助，你可以查看以下官方的Box2D论坛：

www.box2d.org/forum/index.php

你也可以在cocos2d论坛的Physics版面寻求帮助：

www.cocos2diphone.org/forum/forum/7