本页由 六月的流光 翻译自 官方的WIKI

物理特性教程（Physics tutorial）

综述

参看前面的教程基础, 对象创建, 时钟 , 帧层次结构 动画 视口与摄像机 声音与音乐 特效

本教程展示了如何为对象添加物理属性和处理碰撞。

如你所见，物理属性完全是数据驱动的。因此，可以在完全相同的一行代码创建一个拥有物理属性（即通过一个body或没有物理属性的对象。

对象可以动态或静态地链接到一个body（译注：body为Box2D中的一个概念，表示一个物理实体的概念）。每一个body可以由8部分组成。

一个body部分可以定义如下：

- 它的形状（目前可用的只有箱子（ box（译注：长方体）、球体（sphere）和多边形（即 mesh（凸多边形））
- 关于形状大小的信息（箱子的拐角点，球体的球心和半径以及凸多边形的顶点）
- 如果有没指定形状的大小数据，形状会尝试填满整个body（根据对象的大小和缩放）
- “self”标记定义产生碰撞的部分
- “check” 掩码定义了与这部分产生碰撞的其他部分（注释1 同一个物理的两个部分永远不会碰撞）
- 一个标记（Solid）指定了这个形状是否只给出碰撞的信息还是应该影响body的模拟物理运动（弹跳等）
    - 其他的各种属性如弹性（restitution） （摩擦（friction）（密度（density）.....

在本教程中我们创建环绕我们屏幕的静态固体墙。然后我们在中间放置箱子。 要创建的箱子数目是可以通过配置文件调整的，默认为100。

唯一可能的交互操作是使用鼠标的左右键（或者键盘左右方向键）来旋转摄像头。
在旋转时也将会更新我们的模拟的重力矢量。
如此，会让我们有无论摄像头怎么旋转那些箱子都一直往我们屏幕底部掉落的感觉。

我们也可以注册物理事件来为两个碰撞的物体添加可视的特效（FXs）。
默认特效（FX）是一种快速的颜色闪烁，通常也是可以实时调整的（即，重新读取配置文件会使新设置立即生效，因为特效默认是不保存在缓冲中的）。

更新一个对象的缩放比（包括通过FX改变它的缩放比）会更新它的物理属性（即 它的body）。
请注意改变一个有物理特性的body对象代价是很高的，因为我们必须删除当前的形状并按正确的大小重建。这么做是因为我们目前唯一的物理引擎插件是基于Box2D的，它不支持实时重新缩放形状。

本教程只展示了简单的物理特性和碰撞控制，但是，比如说，你也可以获取到对象碰撞和结束碰撞的事件。

详细说明

就像往常一样，我们会以加载配置文件，创建一个时钟并注册我们的更新函数作为开始。

请参考前面的以前的教程以获得更多信息。

我们也创建了我们的墙。实时上我们并不是一个一个的创建，我们把它们在父对象中的一个ChildList组合起来。

```
orxObject_CreateFromConfig("Walls");
```

这看起来我们只是创建了一个叫做Walls的对象，但我们会在配置文件中看到，它实际上是一个会产生一堆墙的容器。

接着，我们创建我们的箱子。

```
for(i = 0; i < orxConfig_GetU32("BoxNumber"); i++)
{
  orxObject_CreateFromConfig("Box");
}
```

如你所见，我们并没有指定任何关于墙或箱子的属性，这些都在配置文件中完成了并且完全是数据驱动的。

然后我们注册到物理事件。

```
orxEvent_AddHandler(orxEVENT_TYPE_PHYSICS, EventHandler);
```

这里没有什么新东西，所以我们直接看下 EventHandler回调函数。

```
if(_pstEvent->eID == orxPHYSICS_EVENT_CONTACT_ADD)
{
  orxOBJECT *pstObject1, *pstObject2;

  pstObject1 = orxOBJECT(_pstEvent->hRecipient);
  pstObject2 = orxOBJECT(_pstEvent->hSender);

  orxObject_AddFX(pstObject1, "Bump");
  orxObject_AddFX(pstObject2, "Bump");
}
```

基本上我们只处理了新的联系事件并且我们增加了一个叫做Bump的FX在两个碰撞的对象上。这个FX会使得它们闪烁一种随机的颜色。
现在看看我们的Update函数。

```
void orxFASTCALL Update(const orxCLOCK_INFO *_pstClockInfo, void
*_pstContext)
{
  orxFLOAT fDeltaRotation = orxFLOAT_0;

  if(orxInput_IsActive("RotateLeft"))
  {
    fDeltaRotation = orx2F(4.0f) * _pstClockInfo->fDT;
  }
  if(orxInput_IsActive("RotateRight"))
  {
    fDeltaRotation = orx2F(-4.0f) * _pstClockInfo->fDT;
  }
```

```
  if(fDeltaRotation != orxFLOAT_0)
  {
    orxVECTOR vGravity;

    orxCamera_SetRotation(pstCamera, orxCamera_GetRotation(pstCamera) +
fDeltaRotation);

    if(orxPhysics_GetGravity(&vGravity))
    {
      orxVector_2DRotate(&vGravity, &vGravity, fDeltaRotation);
      orxPhysics_SetGravity(&vGravity);
    }
  }
}
```

如你所见，我们从RotateLeft和RotateRight输入得到了旋转的更新。
如果需要旋转，那么我们就用orxCamera_SetRotation() 来更新摄像头并且我们更新相应的物理特性模拟的重力（矢量）。
这么做，无论摄像头怎么旋转，我们的箱子总会来看起来是向我们屏幕的底部运动。
注意使用orxVector_2DRotate()以便旋转重力矢量。

注意orx代码中的所有的旋转总是用弧度表示!

现在我们看一下我们的配置数据。你可以在body section of config settingsLINK 配置设置中的body 配置段）找到更多关于配置参数的信息。

首先，我们通过ChildList属性隐式地 [1]创建了许多墙。实现如下：

```
[Walls]
ChildList = Wall1 # Wall2 # Wall3 # Wall4; # Wall5 # Wall6
```

正如我们看到的，我们的墙对象为空，只是创建了 Wall1Wall2Wall3 和Wall4 注意 ‘;’ 在列表的结尾）。你可以移除这个 ‘;’ 新增两堵墙。

现在让我们看看怎么定义我们的墙和它们的物理属性。
让我们从我们将要在WallBody和 BoxBody中都用到的形状开始。

```
[FullBoxPart]
Type        = box
Restitution = 0.0
Friction    = 1.0
SelfFlags   = 0x0001
CheckMask   = 0xFFFF
Solid       = true
Density     = 1.0
```

这里我们创造了一个没有Restitution(弹性）和一点Friction(摩擦)的box (长方体)形状。 同样我们也为这堵墙定义了 SelfFlags 和 CheckMask
第一个（译注SelfFlags定义了标识标记，第二个定义了那些它会碰撞到的标识标记
基本上，如果我们有两个对象Object1和Object2.如果以下条件为TRUE它们就会发生碰撞。

```
(Object1.SeflFlags & Object2.CheckMask) && (Object1.CheckMask &
```

```
Object2.SelfFlags)
```

注意：我们没有为这个FullBoxPart指定 TopLeft 和 BottomRight 属性，所以它会使用所引用的body/object的完整尺寸。

现在我们需要为这些箱子和墙定义我们的body

```
[WallBody]
PartList = FullBoxPart

[BoxBody]
PartList  = FullBoxPart
Dynamic   = true
```

我们可以看到他们都使用相同的部分（注释2 它们最多可以使用8个部分，但这里只使用了一个）。
由于BoxBody箱子的body的Dynamic属性被设置为TRUE这个对象会根据物理特性的模拟移动。
对 WallBody墙的body没有特别指定Dynamic属性（的值），则会默认设置为FALSE并且无论发生什么这些墙都不会移动。
注意：由于两个non-dynamic即 static译注设置Dynamic属性为FALSE或不设置）的对象之间不能发生碰撞，即便两堵墙接触或重叠它们也不会碰撞。

现在我们有了我们的body让我们看看怎么把它们应用到我们的对象。
首先，我们的箱子。

```
[Box]
Graphic   = BoxGraphic
Position  = (50.0, 50.0, 0.0) ~ (750.0, 550.0, 0.0)
Body      = BoxBody
Scale     = 2.0
```

如你所见，我们的箱子有一个被设置为BoxBody的 Body属性。
我们也注意到它的位置随机，这意味着每一次我们创建一个新的箱子，它会有一个在这个范围内的随机位置。

现在看看墙的配置。

```
[WallTemplate]
Body = WallBody

[VerticalWall@WallTemplate]
Graphic = VerticalWallGraphic;
Scale   = @VerticalWallGraphic.Repeat;

[HorizontalWall@WallTemplate]
Graphic = HorizontalWallGraphic;
Scale   = @HorizontalWallGraphic.Repeat;

[Wall1@VerticalWall]
Position = (0, 24, 0)

[Wall2@VerticalWall]
Position = (768, 24, 0)
```

```
[Wall3@HorizontalWall]
Position = (0, -8, 0)

[Wall4@HorizontalWall]
Position = (0, 568, 0)

[Wall5@VerticalWall]
Position = (384, 24, 0)

[Wall6@HorizontalWall]
Position = (0, 284, 0)
```

正如我们看到的我们再一次使用到了继承。
首先我们定义一个包含设置为WallBody的 Body属性的WallTemplate。
然后HorizontalWall和VerticalWall从这个配置段继承。基本上它们除了一个不同的Graphic属性外拥有相同的物理属性。
现在我们有水平和垂直的墙模板了，我们只需要再给它们添加一个位置。
这就是我们对Wall1、Wall2、等……所做的

资源

源代码：08_Physics.c

配置文件：08_Physics.ini

1)
译者注：原文中的implicitely应为implicitly，应该是作者笔误