Box2D v2.3.0 用户手册

Copyright © 2007-2013 Erin Catto 著

Copyright © 2015 Antkillerfarm译

基于

Aman JIANG(江超宇)的v2.0.1用户手册

complex\_ok的2.1.0用户手册

Chapter 1 导言 7

1.1 关于 7

1.2 先决条件 7

1.3 关于手册 7

1.4 反馈和报告BUG 8

1.5 核心概念 8

形状(shape) 8

刚体(rigid body) 8

夹具(fixture) 8

约束(constraint) 8

接触约束(contact constraint) 9

关节(joint) 9

关节限制(joint limit) 9

关节马达(joint motor) 9

世界(world) 9

求解器(solver) 9

连续碰撞(continuous collision) 9

1.6 模块 10

1.7 单位 11

1.8 工厂和定义 11

Chapter 2 Hello Box2D 13

2.1 创建世界 13

2.2 创建地面盒 13

2.3 创建动态物体 14

2.4 模拟(Box2D的)世界 15

2.5 清理 17

2.6 The Testbed 17

Chapter 3 通用模块 19

3.1 关于 19

3.2 配置 19

类型 19

常数 19

分配器包装 19

版本号 19

3.3 内存管理 19

3.4 数学 20

Chapter 4 碰撞模块 21

4.1 关于 21

4.2 形状 21

圆形 21

多边形 22

边框形状（Edge shapes） 24

链接形状（Chain Shapes） 25

4.3 单元几何查询（Unary Geometric Queries） 26

形状点测试（Shape Point Test） 26

形状的光线投射(Shape Ray Cast) 27

4.4 对等函数 27

重叠 27

接触形（Contact Manifolds） 28

距离 29

撞击时间 29

4.5 动态树 30

4.6 Broad-phase 31

Chapter 5 力学模块 33

5.1 概述 33

Chapter 6 物体 34

6.1 关于 34

b2\_staticBody 34

b2\_kinematicBody 34

b2\_dynamicBody 34

6.2 物体定义 34

物体类型 35

位置和角度 35

阻尼 35

Gravity Scale 36

休眠参数 36

固定旋转 36

子弹 37

活动状态 38

用户数据 38

6.3 物体工厂（Body Factory） 38

6.4 Using a Body 39

Mass Data 39

State Information 40

Position and Velocity 40

Chapter 7 Fixtures 41

7.1 About 41

7.2 Fixture Creation 41

Density 42

Friction 42

Restitution 42

Filtering 43

7.3 Sensors 44

Chapter 8 Joints 45

8.1 About 45

8.2 The Joint Definition 45

8.3 Joint Factory 46

8.4 Using Joints 46

8.5 Distance Joint 47

8.6 Revolute Joint 48

8.7 Prismatic Joint 49

8.8 Pulley Joint 50

8.9 Gear Joint 51

8.10 Mouse Joint 52

8.11 Wheel Joint 53

8.12 Weld Joint 53

8.13 Rope Joint 53

8.14 Friction Joint 53

8.15 Motor Joint 54

Chapter 9 Contacts 55

9.1 About 55

contact point 55

contact normal 55

contact separation 55

contact manifold 55

normal impulse 55

tangent impulse 56

contact ids 56

9.2 Contact Class 56

9.3 Accessing Contacts 57

9.4 Contact Listener 57

Begin Contact Event 58

End Contact Event 58

Pre-Solve Event 58

Post-Solve Event 59

9.5 Contact Filtering 62

Chapter 10 World Class 63

About 63

Creating and Destroying a World 63

Using a World 63

Simulation 63

Exploring the World 64

AABB Queries 65

Ray Casts 66

Forces and Impulses 67

Coordinate Transformations 67

Lists 67

Chapter 11 Loose Ends 69

11.1 用户数据 69

Chapter 12 70

12.1 Implicit Destruction 70

12.2 Pixels and Coordinate Systems 71

Chapter 13 Debug Drawing 73

Chapter 14 Limitations 74

Chapter 15 References 75

# 导言

## 关于

Box2D 是一个用于游戏的 2D 刚体仿真库。程序员可以在他们的游戏里使用它，它可以使物体的运动更加真实，并让游戏世界看起来更具交互性。从游戏引擎的视角来看，物理引擎就是一个程序性动画 (proceduralanimation)的系统。

(译注: 做动画常有两种方法, 一种是预先准备好动画所需的数据，比如图片，再一帧一帧地播放。另一种是以一定方法，动态计算出动画所需的数据，根据数据再进行绘图。   
从这种角度看，预先准备的，可称为数据性动画，动态计算的可称为程序性动画。  
这个区别，就类似以前我们做历史题和数学题，做历史题，记忆很重要，也就是答案需要预先准备好的。做数学题，方法就很重要，答案是需要用方法推导出来的。   
Box2D就是用物理学的方法，推导出那游戏世界物体的位置，角度等数据。而Box2D也仅仅推导出数据，至于得到数据之后怎么处理就是程序员自己的事情了。)

Box2D 是用可移植的 C++ 写成的。引擎中大部分类型的定义都有 b2 前缀,希望这能有效的消除它和你的游戏引擎之间的名字冲突。

## 先决条件

在此,我假定你已经熟悉了基本的物理学概念，例如质量、力、扭矩和冲量。如果没有，请先查询一下Google搜索和维基百科。

Box2D是游戏开发者大会(Game Developer Conference，GDC)的物理学教程的一部分。你可以从box2d.org的下载页面获得这些教程。

因为 Box2D 是用 C++ 写成的，所以你应该具备 C++ 程序设计的经验。Box2D 不应该成为你的第一个 C++ 程序项目。你应该已经能熟练地编译，链接和调试了。

**注意**

Box2D 不应该成为你的第一个 C++ 程序项目。你应该已经能熟练地编译，链接和调试了。网络上有很多关于C++的资料。

## 关于手册

这个手册包含了主要的Box2D的API，但并不是每一个都包含了 。你可以通过阅读Box2D自带的testbed程序的代码来学习更多的东西。而且Box2D代码的注释已经按照Doxygen格式编写，可以很容易的创建超链接形式的API文档。

这个手册只在新版本发布的时候更新。因此相比版本库中的代码版本，它可能已经过时了。

## 反馈和报告BUG

如果你有关于Box2D的问题或者反馈意见，请在论坛留言。这也是社区讨论的好地方。

Box2D使用 Google code project来跟踪问题。这是个很好的方式，它可以确保你的问题不会淹没在论坛之中。

请在这里列出BUG和功能需求: <http://code.google.com/p/box2d/>

如果你能提供更有效的细节的话，就能确保你的问题得到解决。一个用于复现问题的testbed用例是很有意义的。你可以在随后章节读到和testbed相关的内容。

## 核心概念

Box2D 中有一些基本的概念和对象，这里我们先做一个简要的定义,在随后的章节里会有更详细的描述。

#### 形状(shape)

形状是一个2D的几何对象。例如圆或多边形。

#### 刚体(rigid body)

一块十分坚硬的物质,它上面的任何两点之间的距离都是完全不变的。它们就像钻石那样坚硬。在后面的讨论中,我们用物体(body)来指代刚体。

#### 夹具(fixture)

夹具将形状绑定到物体上，并添加密度(density)、摩擦(friction)和恢复(restitution)等材料特性。夹具还将形状放入碰撞系统(碰撞检测(Broad Phase))中，以使之能与其他形状相碰撞。

(译注:一个物体和另一物体碰撞, 碰撞后速度和碰撞前速度的比值会保持不变，这比值就叫恢复系数。)

(译注: Broad Phase是碰撞检测的一个子阶段, 将空间分割, 每个空间对应一个子树, 物体就放到树中, 不同子树内的物体不可能相交不用去计算, 在同一个子树由对应的算法再计算出接触点等信息。因为这是远距碰撞检测，就叫Broad Phase, 接下来还有Narrow Phase。)

#### 约束(constraint)

约束(constraint)就是消除物体自由度的物理连接。一个2D物体有3个自由度（两个平移坐标和一个旋转坐标）。如果我们把一个物体钉在墙上(像摆锤那样)，那我们就把它约束到了墙上。这样，此物体就只能绕着这个钉子旋转，因此这个约束消除了它 2 个自由度。

#### 接触约束(contact constraint)

一种防止刚体穿透，并模拟摩擦和恢复的特殊约束。你不必创建接触约束，它们会自动被 Box2D 创建。

#### 关节(joint)

它是一种用于把两个或更多的物体固定到一起的约束。Box2D 支持若干种关节类型: 旋转、棱柱、距离等等。有些关节拥有限制(limits)和马达(motors)。

#### 关节限制(joint limit)

关节限制限定了关节的运动范围。例如，人类的胳膊肘只能做某一范围角度的运动。

#### 关节马达(joint motor)

关节马达能依照关节的自由度来驱动所连接的物体。例如，你可以使用马达来驱动胳膊肘的

旋转。

#### 世界(world)

物理世界就是相互作用的物体，夹具和约束的集合。Box2D 支持创建多个世界，但这通常是不必要或不推荐的。

#### 求解器(solver)

物理世界使用求解器来推算时间，求解接触和关节约束。Box2D的求解器是一种高性能的迭代求解器，它会顺序执行N次，这里的N是约束的个数。

(译注:即算法的复杂度为O（N）。)

#### 连续碰撞(continuous collision)

求解器使用时域上的离散时间步来推算物体状态。如果没有特殊处理的话，这会导致隧穿效应。

(译注：假设我们采用1s的固定时间间隔来推算一个物理系统的运动。那么如果这个系统中有两个物体在某一秒的0.5s的时刻，发生碰撞的话。死板的采用固定时间间隔计算的方法，就会导致物体实际上越过了碰撞点的现象发生，这就是隧穿效应。解决的办法显然是要估算出碰撞发生的时刻，并做相应的处理，这也是下一段提到的TOI的含义。)



Box2D拥有特殊的算法来处理隧穿效应。首先，碰撞算法能够在两个物体的运动过程中进行插值运算，以找到首次碰撞时间 (the first time of impact,TOI)。接着，一个分步求解器将物体移动到它们的TOI时刻，并对碰撞求解。

## 模块

Box2D由三个模块组成: 通用模块(Common)，碰撞模块(Collision) 和力学模块(Dynamics). 通用模块包含了内存分配、数学和配置的代码。碰撞模块定义形状、碰撞检测和碰撞的函数或队列。最终力学模块提供对世界、物体、夹具和关节的模拟。



## 单位

Box2D使用浮点数，所以必须使用公差来保证它正常工作。这些公差已经被调谐得适合米-千克-秒(MKS)单位制。尤其是，Box2D已被调谐得能良好地处理0.1到10米之间的移动物体。这意味着从罐头盒到公共汽车大小的对象都能良好地工作。静态的物体就算大到50米都没有问题。

作为一个2D物理引擎，使用像素作为单位是很诱人的。但很不幸，那将导致不良的模拟，也可能会造成古怪的行为。一个200像素长的物体在Box2D看来就有45层建筑那么大。

**注意**

Box2D 已被调谐至 MKS 单位。移动物体的尺寸应该保持在大约 0.1 到 10 米之间。当你渲染场景和角色时, 可能要用到一些比例缩放系统。Box2D自带的testbed例子，使用了OpenGL的视口变换。不要使用像素！！！

最好把Box2D中的物体看作是被贴上了你的艺术创作品的移动广告板。这个广告板在一个以米为单位的系统里运动，但你可以利用简单的比例因子把它转换为像素坐标。之后就可以使用这些像素坐标去确定你的精灵(sprites)的位置，等等。你也可以将它的坐标轴翻转过来。

(译注:坐标轴翻转的含义是比例因子可以为负数。)

Box2D里的角使用弧度制。物体的旋转角度以弧度方式存储，并可以无限增大。如角度变得太大，可考虑将角度进行规范化。（使用b2Body::SetAngle）

**注意**

Box2D使用弧度，而不是度。

## 工厂和定义

快速内存管理在 Box2D API 的设计中担当了一个中心角色。所以当你创建一个 b2Body 或一个 b2Joint时，你需要调用 b2World 的工厂函数(factory functions)。你不应以别的方式为这些类型分配内存。

这些是创建函数:

b2Body\* b2World::CreateBody(const b2BodyDef\* def)

b2Joint\* b2World::CreateJoint(const b2JointDef\* def)

这些是对应的销毁函数:

void b2World::DestroyBody(b2Body\* body)

void b2World::DestroyJoint(b2Joint\* joint)

当你创建物体或关节时，需要提供定义(definition)。这些定义包含了创建物体或关节时所需的所有信息。使用这样的方法，我们能够预防构造错误，保持较少的函数参数数量，提供有意义的默认值，并减少访问子(accessor)的个数。

因为fixture必须有父body，所以要使用b2Body的工厂方法来创建并销毁它们。

b2Fixture\* b2Body::CreateFixture(const b2FixtureDef\* def)

void b2Body::DestroyFixture(b2Fixture\* fixture)

也有个简便的方法直接用形状和密度来创建fixture。

b2Fixture\* b2Body::CreateFixture(const b2Shape\* shape, float32 density)

工厂并不保留定义的引用，因此你可以在栈上创建定义，并在临时资源中保存它们。

# Hello Box2D

Box2D的发布包中有个Hello World程序。程序创建了一个大大的地面盒(ground box)和一个小小的动态盒(dynamic box)。代码没有涉及到图形界面，你只能在控制台中看到随时间变化的盒子位置的文字输出。

这是个很好的例子, 展示了怎么学习和使用Box2D。

## 创建世界

每个Box2D程序开始时都会创建一个b2World对象。b2World是个物理枢纽(physics hub)，用于管理内存、对象和模拟。你可以在栈, 堆或数据区中创建出world。

创建Box2D的world很简单。首先，我们定义重力矢量。

b2Vec2 gravity(0.0f, -10.0f);

现在可以创建world对象了。注意，我们是在栈中创建world, 所以world不能离开它的作用域。

b2World world(gravity);

现在我们已经有了自己的物理世界，开始向里面加东西了。

## 创建地面盒

body用以下步骤来创建：

1. 用位置(position), 阻尼(damping)等来定义body。
2. 用world对象来创建body。
3. 用形状(shape), 摩擦(friction), 密度(density)等来定义fixture。
4. 在body上来创建fixture。

第一步，创建ground body。为此我们需要一个body定义。在定义中，我们指定ground body的初始位置。

b2BodyDef groundBodyDef;

groundBodyDef.position.Set(0.0f, -10.0f);

第二步，将body定义传给world对象，用以创建ground body。world对象并不保留body定义的引用。body默认是静态的。静态物体和其它静态物体之间并没有碰撞，它们是固定的。

b2Body\* groundBody = world.CreateBody(&groundBodyDef);

第三步，创建地面多边形。我们用简便函数SetAsBox使得地面多边形构成一个盒子形状，盒子的中心点就是父body的原点。

b2PolygonShape groundBox;

groundBox.SetAsBox(50.0f, 10.0f);

SetAsBox函数接收半个宽度和半个高度作为参数。因此在这种情况下，地面盒就是100个单位宽(x轴),20个单位高(y轴)。Box2D已被调谐 到使用米，千克和秒做单位。你可以认为长度单位就是米。当物体的大小跟真实世界一样时，Box2D通常工作良好。例如，一个桶约1米高。由于浮点算法的局 限性，使用Box2D模拟冰川或沙尘的运动并不是一个好主意。

第四步，我们创建shape fixture，以完成ground body。在这步中，我们有个简便方法。我们并不需要修改fixture默认的材质属性，可以直接将形状传给body而不需要创建fixture的定义。随后的教程中，我们将会看到如何使用fixture定义来定制材质属性。第二个参数是形状密度，单位是千克/平方米。静态物体的质量定义为0，因此密度对它们是没有用的。

groundBody->CreateFixture(&groundBox, 0.0f);

Box2D并不保存shape的引用。它把数据复制到一个新的b2Shape对象中。

注意，每个fixture都必须有一个父body，即使fixture是静态的。然而，你可以把所有的静态fixture都依附在单个静态body之上。

当你使用fixture 向body添加shape的时候，shape的坐标对于body来说就变成本地的了。因此当body移动的时候，shape也一起移动。fixture的世界变换继承自它的父body。fixture没有独立于body的变换。所以我们不需要移动body上的shape。不支持移动或修改body上的shape。原因很简单：形状发生改变的物体不是刚体，而Box2D只是个刚体引擎。Box2D所做的很多假设都是基于刚体模型的。如果这一条被改变的话，很多事情都会出错。

## 创建动态物体

现在我们已经有了一个地面body，我们可以使用同样的方法来创建一个动态body。除尺寸之外的主要区别是，我们必须为动态body设置质量属性。

首先我们用CreateBody创建body。默认情况下，body是静态的，所以在构造时候应该设置b2BodyType，使得body成为动态的。

b2BodyDef bodyDef;

bodyDef.type = b2\_dynamicBody;

bodyDef.position.Set(0.0f, 4.0f);

b2Body\* body = world.CreateBody(&bodyDef);

注意

如果你想让body受力的影响而运动, 你必须将body的类型设为b2\_dynamicBody。

然后，我们创建一个多边形shape, 并将它附加到fixture定义上。我们先创建一个box shape：

b2PolygonShape dynamicBox;

dynamicBox.SetAsBox(1.0f, 1.0f);

接下来，我们使用box创建一个fixture定义。注意, 我们把密度值设置为1，而密度值默认是0。并且，shape的摩擦系数设置为0.3。

b2FixtureDef fixtureDef;

fixtureDef.shape = &dynamicBox;

fixtureDef.density = 1.0f;

fixtureDef.friction = 0.3f;

注意

一个动态body至少有一个密度不为0的fixture。否则会出现一些奇怪的行为。

使用fixture定义，我们现在就可以创建fixture。这会自动更新body的质量。要是你喜欢，你可以为body添加许多不同的fixture。每个fixture都会增加物体的总质量。

body->CreateFixture(&fixtureDef);

这就是初始化过程。现在我们已经做好准备，可以开始模拟了。

## 模拟(Box2D的)世界

我们已经初始化了地面box和一个动态box。该让牛顿来接手了。我们只有少数几个问题需要考虑。

Box2D使用了一种名叫积分器(integrator)的数值算法。 积分器在离散的时间点上模拟物理方程。 它与传统的游戏动画循环一同运行。我们需要为Box2D选取一个时间步（time step）。通常来说用于游戏的物理引擎需要至少 60Hz 的速度，也就是 1/60 秒的时间步。你可以使用更大的时间步，但是你必须更加小心地为你的世界调整定义。我们也不喜欢时间步变化得太大，一个变化的时间步会导致变化的结果，这会给调试带来困难。所以不要把时间步关联到帧频(除非你真的必须这样做)。直截了当地,这个就是时间步。

float32 timeStep = 1.0f / 60.0f;

除积分器外,Box2D代码还使用了约束求解器(constraint solver)。约束求解器用于解决模拟中的所有约束，一次一个。单个的约束会被完美的求解，然而当我们求解一个约束的时候，我们就会稍微干扰另一个约束。要得到良好的解，我们需要多次迭代所有约束。

约束求解有两个阶段：速度阶段和位置阶段。在速度阶段，求解器会计算必要的冲量，使得物体正确运动。而在位置阶段，求解器会调整物体的位置，减少物体之间的重叠和关节的脱节。每个阶段都有自己的迭代计数。此外，如果误差已足够小的话，位置阶段的迭代可能会提前退出。

Box2D建议的迭代次数，对于速度是8次，对于位置是3次。你可以按自己的喜好去调整这个数字，但要记得它是性能与精度之间的折中。更少的迭代会增加性能但降低精度，同样地，更多的迭代会降低性能但能提高模拟的质量。对于这个简单示例，我们不需要很多的迭代。这里是我们选择的迭代次数。

int32 velocityIterations = 6;

int32 positionIterations = 2;

注意，时间步和迭代数是完全无关的。一个迭代并不是一个子步。一次迭代就是在时间步之中的单次遍历所有约束，你可以在单个时间步内多次遍历约束。

现在我们可以开始模拟循环了，在你的游戏中，模拟循环和游戏循环可以合并起来。每次游戏循环你都应该调用b2World::Step，通常调用一次就够了，这取决于帧频以及物理时间步。

这个Hello World程序设计得非常简单，它没有图形输出。代码会打印出动态body的位置以及旋转角。这就是模拟 1 秒钟内 60 个时间步的循环。

for (int32 i = 0; i < 60; ++i)

{

world.Step(timeStep, velocityIterations, positionIterations);

b2Vec2 position = body->GetPosition();

float32 angle = body->GetAngle();

printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle);

}

输出展示了box下降并降落到地面的情况。你的输出看起来应当是这样的：

0.00 4.00 0.00

0.00 3.99 0.00

0.00 3.98 0.00

...

0.00 1.25 0.00

0.00 1.13 0.00

0.00 1.01 0.00

## 清理

当world对象超出它的作用域，或通过指针将其 delete 时，分配给body、 fixture和 joint使用的内存都会被释放。这能提升性能，并使你的生活变得更简单。然而，你应该将body、fixture或joint的指针都清零，因为它们已经无效了。

## The Testbed

一旦你征服了 HelloWorld 例子，你应该开始看 Box2D 的 testbed 了。testbed 是个单元测试框架，也是个演示环境，这是它的一些特点：

* 可移动和缩放的摄像机。
* 可用鼠标选中依附在动态物体上的形状。
* 可扩展的测试集。
* 通过图形界面选择测试,调整参数,以及设置调试绘图。
* 暂停和单步模拟。
* 文字渲染。



在 testbed 中有许多 Box2D 的测试用例，以及框架本身的实例。我鼓励你通过研究和修改它来学习 Box2D。

注意:testbed 是使用 freeglut 和 GLUI 写成的，testbed 本身并不是 Box2D 库的一部分。Box2D本身并不知道如何渲染。就像 HelloWorld 例子一样，使用 Box2D 并不一定需要渲染。

# 通用模块

## 关于

通用模块包含了配置(Settings)，内存管理(memory management)和矢量数学(vector math)。

## 配置

头文件 b2Settings.h 包含：

* 类型，比如int32和float32
* 常数
* 分配器包装(Allocation wrappers)
* 版本号

### 类型

Box2D定义了不同的类型，比如float32、 int8等，以方便确定结构的大小。

### 常数

Box2D定义了若干常数，这些都记录在b2Settings.h中。通常情况下，你不需要调整这些常数。

Box2D的碰撞计算和物体模拟使用了浮点数学。考虑到有舍入方面的错误，所以要定义一些数值公差的，一些是绝对公差，另一些是相对公差。绝对公差使用MKS单位。

### 分配器包装

配置文件定义了b2Alloc和b2Free，用于大内存的分配。你可以让b2Alloc和b2Free调用你自己的内存管理系统。

### 版本号

b2Version结构保存了当前的版本信息，你可以在运行时(run-time)查询。

## 内存管理

Box2D 设计上的很多决定，都是为了能快速有效地使用内存。在本节我将论述 Box2D 如何及为什么要这样分配内存。

Box2D 倾向于分配大量的小型对象(50-300 字节左右)。在系统的堆(heap)上，通过 malloc 或 new分配内存既低效，又容易产生内存碎片。大多数小型对象的生命期都很短暂，例如触点 (contact)，可能只会维持几个时间步。所以我们需要为这些对象提供一个有效的堆分配器。

Box2D的解决方案是使用名为b2BlockAllocator的小型对象分配器(SOA)。SOA维护了一些不定尺寸并可扩展的内存池。当有内存分配请求时，SOA 会返回一块大小最匹配的内存。当内存块释放之后，它会被回收到池中。这些操作都十分快速，只有很小的堆流量。

因为 Box2D 使用了SOA，所以你永远也不应该去 new 或 malloc 一个body、fixture或joint。你只需分配一个 b2World，它为你提供了创建body、fixture和joint的工厂(factory)。这使得Box2D可以使用 SOA 并且将具体的细节隐藏起来。同样，绝对不要去 delete 或 free 一个body、fixture或joint。

当执行一个时间步的时候，Box2D 会需要一些临时的内存。为此，它使用了一个栈分配器来消除单步的堆分配，这个分配器命名为b2StackAllocator。你不需要关心栈分配器，但对此有所了解还是不错的。

## 数学

Box2D包含了一个简单精细的矢量和矩阵模块，来满足Box2D内部和API接口的需要。所有的类都是公开的，你可以在自己的应用程序中自由使用它们。

数学库保持得尽量简单, 使得Box2D容易移植和维护。

# 碰撞模块

## 关于

碰撞模块包含了形状和操作形状的函数。该模块还包含了动态树(dynamic tree)和broad-phase，用于加快大型系统的碰撞处理速度。

碰撞模块被设计为可用于动态系统之外的地方。例如，你可以将动态树用于你的游戏中，除了物理之外的目的。

然而，Box2D 的主要目标是提供一个刚体物理引擎。因此对于一些应用来说，使用碰撞模块会感觉受到限制。同样的，我也不是很想将之写成文档，并形成API。

## 形状

形状描述了可相互碰撞的几何对象，它的使用独立于物理模拟。最起码，你应该知道如何创建shape，并将之附加到刚体上。

b2Shape是个基类，Box2D的各种形状都实现了这个基类。此基类定义了几个函数：

* 判断一个点与形状是否有重叠。
* 在形状上执行光线投射(ray cast)。
* 计算形状的AABB。
* 计算形状的质量。

另外,，每个形状都有成员变量：类型(type)和半径(radius)。 对于多边形，半径也是有意义的，下面会进行讨论。

需要注意的是shape并不知道body，也与力学系统无关。Shape采用一种紧凑格式来进行存储，这种格式经过尺寸和性能的优化。因此，shape并不方便移动，你必须通过手动的设置形状顶点来移动shape。然而，当使用fixture将shape添加到body上之后，shape就会和他的宿主body一起移动。总之：

* 当一个shape**没有**添加到body上时，它的顶点用世界坐标系来表示。
* 当一个shape添加到body上时，它的顶点用局部坐标系来表示。

### 圆形

圆形有位置和半径。圆形是实心的，你没有办法使圆形变成空心。

b2CircleShape circle;

circle.m\_p.Set(2.0f, 3.0f);

circle.m\_radius = 0.5f;

### 多边形

Box2D的多边形是实心的凸(Convex)多边形。在多边形内部任意选择两点，作一线段，如果所有的线段跟多边形的边都不相交，这个多边形就是凸多边形。多边形是实心的，而不是空心的。一个多边形必须有3个或以上的顶点。



多边形的顶点以逆时针（counter clockwise winding，CCW）的顺序存储。我们必须很小心，逆时针是相对于右手坐标系统来说的，这坐标系下，Z轴指向平面外面。有可能相对于你的屏幕，就变成顺时针了，这取决于你自己的坐标系统是怎么规定的。



多边形的成员变量具有public访问权限，但是你应该使用初始化函数来创建多边形。初始化函数会创建法向量(normal vectors)，并检查参数的合法性。

创建多边形时，你可以传递一个包含顶点的数组。数组大小最多是b2\_maxPolygonVertices，这数值默认是8。这已足够描述大多数的凸多边形了。

b2PolygonShape::Set 函数会自动计算凸包（convex hull），并建立时针序。这个函数在顶点数少的时候，是非常快的。但如果你增大b2\_maxPolygonVertices ，凸包的计算就会变慢。此外，凸包函数会消除你提供的顶点，或者对其重排序。距离小于b2\_linearSlop 的顶点会被合并。

// This defines a triangle in CCW order.

b2Vec2 vertices[3];

vertices[0].Set(0.0f, 0.0f);

vertices[1].Set(1.0f, 0.0f);

vertices[2].Set(0.0f, 1.0f);

int32 count = 3;

b2PolygonShape polygon;

polygon.Set(vertices, count);

多边形有一些方便的函数来创建box。

void SetAsBox(float32 hx, float32 hy);

void SetAsBox(float32 hx, float32 hy, const b2Vec2& center, float32 angle);

多边形从b2Shape中继承了半径。通过半径，在多边形的周围创建了一个保护层(skin)。堆叠的情况下，此保护层让多边形之间保持稍微分开。这使得可以在核心多边形上执行连续碰撞。



多边形保护层通过保持多边形的分离来防止隧穿效应。这会导致形状之间有小空隙。你的显示可以比多边形大些，来隐藏这些空隙。



### 边框形状（Edge shapes）

边框形状由一些线段组成。它们可辅助为你的游戏创建一个形状自由的静态环境。边框形状的主要限制在于它们能够与圆形和多边形碰撞，但它们之间却不会碰撞。Box2D使用的碰撞算法要求两个碰撞物体中至少有一个有体积。边框形状没有体积。所以边框形状之间的碰撞是不可能的。

// This an edge shape.

b2Vec2 v1(0.0f, 0.0f);

b2Vec2 v2(1.0f, 0.0f);

b2EdgeShape edge;

edge.Set(v1, v2);

在很多情况下，游戏环境是若干个边框形状首尾相连构成的。当多边形沿着边框链滑动的时候，会导致一个异常的行为。在下图中，我们可以看到一个box和一个内部顶点之间的碰撞。当多边形和一个内部顶点碰撞时，会产生一个内部碰撞法线，这会导致“幽灵”碰撞现象。



如果edge1不存在的话，这个碰撞看起来还算正常。edge1存在的话，这个内部碰撞就像是Bug了。但是通常当Box2D处理两个形状的碰撞时，会单独处理他们的。

幸运的是，边框形状提供了一种机制来消除幽灵碰撞——存储用于调整的幽灵顶点。Box2D使用这些幽灵顶点来阻止内部碰撞。



// This is an edge shape with ghost vertices.

b2Vec2 v0(1.7f, 0.0f);

b2Vec2 v1(1.0f, 0.25f);

b2Vec2 v2(0.0f, 0.0f);

b2Vec2 v3(-1.7f, 0.4f);

b2EdgeShape edge;

edge.Set(v1, v2);

edge.m\_hasVertex0 = true;

edge.m\_hasVertex3 = true;

edge.m\_vertex0 = v0;

edge.m\_vertex3 = v3;

通常情况下，用这种方式将边框缝合到一起是有些浪费和无聊的。这就为我们引入了链接形状（chain shapes）的概念。

### 链接形状（Chain Shapes）

链接形状提供了一种有效的方式，来将许多边框连接在一起，用以构建你的静态游戏世界。链接形状自动消除幽灵碰撞，并提供两侧的碰撞。



// This a chain shape with isolated vertices

b2Vec2 vs[4];

vs[0].Set(1.7f, 0.0f);

vs[1].Set(1.0f, 0.25f);

vs[2].Set(0.0f, 0.0f);

vs[3].Set(-1.7f, 0.4f);

b2ChainShape chain;

chain.CreateChain(vs, 4);

你可能会有一个滚动的游戏世界，需要将若干个链接连接到一起。你可以用幽灵顶点来连接链接，就像我们在b2EdgeShape 中所做的那样。

// Install ghost vertices

chain.SetPrevVertex(b2Vec2(3.0f, 1.0f));

chain.SetNextVertex(b2Vec2(-2.0f, 0.0f));

你也可以自动创建一个环。

// Create a loop. The first and last vertices are connected.

b2ChainShape chain;

chain.CreateLoop(vs, 4);

不支持自相交的链接形状。它可能正常工作，也可能不会。防止幽灵碰撞的代码假定链接中没有自相交存在。同样的，非常接近的顶点也会导致问题。需要确保你的所有边框都比b2\_linearSlop (5mm)长。



链接中的每个边框都被认为是一个子形状，并能用索引来访问。当一个链接形状被连到body上时，每个边框在碰撞检测树上都会有它自己的包围盒（bounding box）。

// Visit each child edge.

for (int32 i = 0; i < chain.GetChildCount(); ++i)

{

b2EdgeShape edge;

chain.GetChildEdge(&edge, i);

…

}

## 单元几何查询（Unary Geometric Queries）

你可以在一个单独的形状上，执行一系列的几何查询。

### 形状点测试（Shape Point Test）

你可以测试一个点是否与形状有所重叠。你需要提供一个形状的变换以及世界坐标上的一个点。

b2Transfrom transform;

transform.SetIdentity();

b2Vec2 point(5.0f, 2.0f);

bool hit = shape->TestPoint(transform, point);

边框和链接形状总是返回false，即使链接是一个环。

### 形状的光线投射(Shape Ray Cast)

你可以用光线射向形状，得到它们之间的第一个交点和法向量。如果在形状内部开始投射，就当成没有交点。链接形状包含儿子索引，是因为光线投射一次只会检测一个边框。

b2Transfrom transform;

transform.SetIdentity();

b2RayCastInput input;

input.p1.Set(0.0f, 0.0f, 0.0f);

input.p2.Set(1.0f, 0.0f, 0.0f);

input.maxFraction = 1.0f;

int32 childIndex = 0;

b2RayCastOutput output;

bool hit = shape->RayCast(&output, input, transform, childIndex);

if (hit)

{

b2Vec2 hitPoint = input.p1 + output.fraction \* (input.p2 – input.p1);

…

}

## 对等函数

碰撞模块包含一些对等函数，它们接受一对形状参数，并计算出结果。这些函数包括：

* 重叠
* 接触形
* 距离
* 撞击时间

### 重叠

你可以用这个函数测试两个形状是否重叠。

b2Transform xfA = …, xfB = …;

bool overlap = b2TestOverlap(shapeA, indexA, shapeB, indexB, xfA, xfB);

如果是链接形状的话，你还必须提供子形状的索引。

### 接触形（Contact Manifolds）

Box2D有一些用来计算重合形状之间的接触点的函数。考虑一下圆与圆，圆与多边形的碰撞，我们只会得到一个接触点和一个向量。多边形与多边形的碰撞，我们可 以得到两个接触点。这些接触点具有相同的法向量，所以Box2D将它们归成一组，构成manifold结构。接触求解器将利用这个结构，以改善堆叠的稳定性。



通常你不需要直接计算接触形，但你可能会使用在模拟过程中已处理好的结果。

b2Manifold结构含有一个法向量和最多两个接触点。向量和接触点都是相对于局部坐标系。为方便接触求解器处理，每个接触点都存储了法向冲量和切向(摩擦）冲量。

存储在b2WorldManifold结构中的数据为内部使用做了优化。如果你需要这些数据，最好的方法是使用b2WorldManifold结构生成世界坐标下的接触向量和点。你需要提供b2Manifold结构和形状的转换及半径。

b2WorldManifold worldManifold;

worldManifold.Initialize(&manifold, transformA, shapeA.m\_radius,

transformB, shapeB.m\_radius);

for (int32 i = 0; i < manifold.pointCount; ++i)

{

b2Vec2 point = worldManifold.points[i];

…

}

注意worldmanifold使用的点数量来自于manifold.

模拟过程中，形状会移动而manifold可能会改变。接触点有可能会添加或移除。你可以使用b2GetPointStates来检查状态。

b2PointState state1[2], state2[2];

b2GetPointStates(state1, state2, &manifold1, &manifold2);

if (state1[0] == b2\_removeState)

{

// process event

}

### 距离

b2Distance函数可以用来计算两个形状之间的距离。距离函数需要两个形状都被转成b2DistanceProxy。重复调用距离函数的时候，Box2D会使用缓冲的方式使之热启动。你可以在b2Distance.h 中看到实现的细节。

(译注：热启动是相对于冷启动而言的。一般情况下，机器热启动时，由于初始化的东西较少，而具有更快的速度。这里将机器热启动的概念，拓展推广到函数热启动，含义都是类似的。)



### 撞击时间

如果两个形状快速移动，它们可能会在一个时间步内穿过对方。



b2TimeOfImpact函数用于确定两个形状运动时碰撞的时间。这称为撞击时间(time of impact, TOI)。b2TimeOfImpact的主要目的是防止隧穿效应。特别是，它设计来防止运动的物体隧穿过静态的几何形状。

这个函数考虑了形状的旋转和平移，但如果旋转足够大，这函数还是会错过碰撞。函数仍然会报告一个非重叠的时间，并捕捉到所有的平移碰撞。

撞击时间函数定义了一条初始的分离轴，并确保形状没有越过这条轴。这可能会在最终位置错过一些碰撞。尽管如此，这种方法在防止隧穿方面已经快速并足够适用了。





很难去限定旋转角的范围，有些情况下，就算是很小的旋转角也会导致错过碰撞。通常，就算错过了一些碰撞，也不会影响到游戏的好玩性。游戏往往会忽略这些碰撞。

这函数需要两个形状(转换成b2DistanceProxy)和两个b2Sweep结构。b2Sweep结构定义了形状的开始和结束时的转换。

你可以在固定旋转角的情况下去执行这个计算撞击时间的函数，这样就不会错过任何碰撞。

## 动态树

Box2D使用b2DynamicTree来高效地组织大量的形状。这个类并不知道形状的存在。取而代之，它通过用户数据指针来操作轴对齐包围框(AABB)。

动态树是分层的AABB树。树的每个内部节点都有两个子节点。叶子节点是用户的AABB。这个树使用旋转来保持树的平衡，即使是在退化的输入（degenerate input）的情况下。

(译注：degenerate input在算法上是指最坏情况的输入。例如要对一个数组进行升序排列。通常情况下的输入，一般是随机的数字序列。而一个降序的数字序列在这里就算是degenerate input了。)

这种树结构支持高效的光线投射(ray casts)和区域查询(region queries)。比如，场景中有数百个形状，你想对场景执行光线投射，如果采用蛮力，就需要对每个形状都进行投射。这是很低效的，并没有利用到形状的分布信息。替代方法是，你维护一棵动态树，并对树进行光线投射。在遍历树的时候，可以跳过大量的形状。

区域查询使用树来查找跟需查询的AABB有重叠的所有叶节点。这比蛮力算法高效得多，因为很多形状会被直接跳过。





通常你并不会直接用到动态树。你会通过b2World类来执行光线投射和区域查询。如果你想创建自己的动态树，你可以去看看Box2D是怎么使用动态树的。

## Broad-phase

物理步内的碰撞处理可以分成两个阶段: narrow-phase和broad-phase。narrow-phase时，我们去计算两个形状之间的接触点。假设有N个形状，使用蛮力算法的话，就需要执行 N\*N/2次narrow-phase。

Tb2BroadPhase类使用了动态树来减少管理数据方面的开销。这可以大幅度减少narrow-phase的调用次数。

通常你不会直接和broad-phase交互。Box2D自己会在内部创建并管理broad-phase。另外要注意，b2BroadPhase是设计用于Box2D中的物理模拟，它可能不适合处理其它情况。

# 力学模块

## 概述

力学模块是Box2D中最复杂的部分，也是与你交互最多的部分。力学模块构建在通用和碰撞模块的基础上，到现在你对这两个模块也应该有所了解了。

力学模块包括下面这些类：

* 夹具
* 刚体
* 接触
* 关节
* 世界
* 监听者

这些类相互依赖，很难在不提及其它类的情况下单独描述一个类。在接下来的章节中，你会看到一些类是之前没有提及过的。你可以先快速浏览一下对应的章节，之后才去细读。

力学模块包括接下来的章节。

# 物体

## 关于

物体具有位置和速度。你可以将力(forces)、扭矩(torques)、冲量(impulses)应用到物体上。 物体可以是静态的(static)、运动但不受力的(kinematic)或动态的(dynamic)。这是物体的类型定义：

#### b2\_staticBody

static物体在模拟时不会运动，就好像它具有无穷大的质量。在Box2D内部，会将static物体的质量和质量的倒数存储为零。static物体可以让用户手动移动。它的速度为零，另外也不会和其它static或kinematic物体相互碰撞。

#### b2\_kinematicBody

kinematic物体在模拟时以一定的速度运动，但不受力的作用。它们可以让用户手动移动，但通常的做法是设置一定的速度来移动它。kinematic物体的行为表现就好像它具有无穷大的质量，Box2D将它的质量和质量的倒数存储为零。

#### b2\_dynamicBody

dynamic物体被完全模拟。它们可以让用户手动移动，但通常它们都是受力的作用而运动。dynamic物体可以和其它所有类型的物体相互碰撞。dynamic物体的质量总是有限大的，非零的。如果你试图将它的质量设置为零，它会自动地将质量修改成一千克，并且它不会转动。

物体是fixtures的骨架，带着fixture在世界中运动。Box2D中的物体总是刚体(rigid body)。也就是说，同一物体上的两个fixture，永远不会相对移动，也不会碰撞。

fixture有可碰撞的几何形状和密度(density)。物体通常从它的fixture中获得质量属性。当物体构建之后，你也可以改写它的质量属性。

通常你会保存所有你所创建物体的指针，这样你就能查询物体的位置，用于更新图形实体的位置。另外在不需要它们的时候，你也可以使用指针去摧毁它们。

## 物体定义

在创建物体之前你需要先创建物体定义(b2BodyDef)。物体定义含有创建并初始化物体所需的数据。

Box2D会从物体定义中复制数据，并不会保存它的指针。这意味着你可以重复使用同一个物体定义去创建多个物体。

让我们看一些物体定义的关键成员。

### 物体类型

本章开始已经说过，有三种物体类型: static、kinematic和dynamic。 你应该在创建时就确定好物体类型，因为以后再修改的话，代价会很高。

bodyDef.type = b2\_dynamicBody;

物体类型是一定要设置的。

### 位置和角度

物体定义为你提供了一个在创建时初始化位置的机会。这比在world原点下创建物体后再移动到某个位置更高效。

**注意**

不要在原点创建物体后再移动它。如果你在原点上同时创建了几个物体，性能会很差。

物体上主要有两个让人感兴趣的点。第一个是物体的原点。fixture和关节都是相对于原点而依附到物体上面的。第二个是物体的质心。质心由形状的质量分布决定，或显式地由b2MassData设置。Box2D内部许多计算都要使用物体的质心, 例如b2Body会存储质心的线速度。

当你构造物体定义的时候,可能你并不知道质心在哪里。你可以指定物体的原点，也可以以弧度指定物体的角度，角度并不受质心位置的影响。如果随后你改变了物体的质量属性，那么质心也会随之移动，但是原点不会改变，物体上的形状和关节也不会移动。

bodyDef.position.Set(0.0f, 2.0f); // the body's origin position.

bodyDef.angle = 0.25f \* b2\_pi; // the body's angle in radians.

刚体也是个参考框架。你可以在这个框架内定义fixture和joint。fixture和joint的锚点不会在框架内移动。

### 阻尼

阻尼用于减小物体在世界中的速度。阻尼跟摩擦有所不同，摩擦仅在物体有接触的时候才会发生。阻尼并不能取代摩擦，往往这两个效果需要同时使用。

阻尼参数的范围可以在0到无穷大之间，0表示没有阻尼，无穷大表示满阻尼。通常来说，阻尼的值应 该在0到0.1之间。通常我不使用线性阻尼， 因为它会使物体看起来有点漂浮。

bodyDef.linearDamping = 0.0f;

bodyDef.angularDamping = 0.01f;

阻尼类似稳定性与性能， 在值较小的时候阻尼效应几乎不依赖于时间步，值较大的时候阻尼效应将随着时间步而变化。如果你使用固定的时间步(推荐)这就不是问题了。

### 重力因子

你可以使用重力因子来调整单个物体上的重力。这需要足够的细心，增加的重力会降低稳定性。

// Set the gravity scale to zero so this body will float

bodyDef.gravityScale = 0.0f;

### 休眠参数

休眠是什么意思？模拟物体的成本是高昂的，所以物体越少，那模拟的效果就越好。当物体停止了运动时，我们会希望停止模拟它。

当Box2D确定一个物体(或一组物体)已停止移动时，物体就会进入休眠状态。休眠物体只消耗很小的 CPU开销。如果一个醒着的物体接触到了一个休眠中的物体，那么休眠中的物体就会醒过来。当物体上的关节或触点被摧毁的时候，它们同样会醒过来。你也可以手动地唤醒物体。

通过物体定义,你可以指定一个物体是否可以休眠，或者创建一个休眠的物体。

bodyDef.allowSleep = true;

bodyDef.awake = true;

### 固定旋转

你可能想让一个刚体，比如某个角色，具有固定的旋转角。这样物体即使在负载下，也不会旋转。 你可以设置fixedRotation来达到这个目的：

bodyDef.fixedRotation = true;

固定旋转标记使得转动惯量和它的倒数被设置成零。

### 子弹

游戏模拟通常以一定帧率(frame rate)产生一系列的图片。这就是所谓的离散模拟。在离散模拟中，在一个时间步内刚体可能移动较大距离。如果一个物理引擎没有处理好大幅度的运动，你就可能会看见一些物体错误地穿过了彼此。这被称为隧穿效应(tunneling)。

默认情况下，Box2D会通过连续碰撞检测(CCD)来防止动态物体穿越静态物体。这是通过扫描形状从旧位置到新位置的过程来完成的。引擎会查找扫描中的新碰撞，并为这些碰撞计算碰撞时间 (TOI)。物体会先被移动到它们的第一个TOI，然后求解器执行一个子步（sub-step）计算以完成整个时间步的计算。在子步中，可能还有更多的TOI事件发生。

一般情况下，dynamic物体之间不会应用CCD，这是为了保持合理的性能。在一些游戏场景中，你需要在动态物体上也使用CCD。比如，你可能想用一颗高速的子弹去射击一块动态的砖头。没有CCD，子弹就可能会隧穿砖头。

在Box2D中，高速移动的物体可以标记成子弹(bullet)。子弹跟static或者dynamic物体之间都会执行CCD。你需要按照游戏的设计来决定哪些物体是子弹。如果你决定一个物体应该按照子弹去处理，可使用下面的设置。

bodyDef.bullet = true;

子弹标记只影响dynamic物体。

### 活动状态

你可能希望创建一个物体并不参与碰撞和动态模拟。这状态跟休眠有点类似，但并不会被其它物体唤醒，它上面的fixture也不会 被放到broad-phase中。也就是说，物体不会参于碰撞检测，光线投射(ray casts)等等。

你可以创建一个非活动的物体，之后再激活它。

bodyDef.active = true;

关节也可以连接到非活动的物体。但这些关节并不会被模拟。你要小心，当激活物体时，它的关节不会被扭曲(distorted)。

注意，激活一个物体和重新创建一个物体的开销差不多。因此你不应该在流世界（streaming worlds）中使用激活，而应该用创建和销毁来节省内存。

(译注：streaming worlds是指该世界中的大多数物体是动态创建的，而不是一开始就有的。)

### 用户数据

用户数据是个void指针。它让你将物体和你的应用程序关联起来。你应该保持一致性，所有物体的用户数据都指向相同的对象类型。

b2BodyDef bodyDef;

bodyDef.userData = &myActor;

## 物体工厂（Body Factory）

world类提供物体工厂来创建和摧毁物体。这让world可以通过一个高效的分配器来创建物体，并且把物体添加到world的数据结构中。

b2Body\* dynamicBody = myWorld->CreateBody(&bodyDef);

... do stuff ...

myWorld->DestroyBody(dynamicBody);

dynamicBody = NULL;

**注意**

永远不要使用new或malloc来创建物体，否则世界不会知道这个物体的存在，并且物体也不会被适当地初始化。

Box2D并不保存物体定义的引用，也不保存其任何数据(除了用户数据指针)。所以你可以创建临时的物体定义，并重复利用它。

Box2D允许你通过删除b2World对象来摧毁物体，它会为你做所有的清理工作。然而，你必须小心地将保存在游戏引擎中的body指针清零。

当你摧毁物体时，依附其上的fixture和joint都会自动被摧毁。了解这点，对你如何管理shape 和joint指针有重要意义。

## 使用物体

在创建完一个物体之后，你可以对它进行许多操作。其中包括设置质量属性，访问其位置和速度，施加力，以及转换点和向量。

### 质量数据

每个物体都有质量（标量）、质心（二维向量）和转动惯性（标量）。对于static物体，它的质量和转动惯性都被设为零。当物体设置成固定旋转(fixed rotation)，它的转动惯性也是零。

通常情况下，当fixture添加到物体上时，物体的质量属性会自动地确定。你也可以在运行时(run-time)调整物体的质量。当你有特殊的游戏方案需要改变质量时，可以这样做。

void SetMassData(const b2MassData\* data);

直接设置物体的质量后，你可能希望再次使用fixture所展示的质量。你可以这样做：

void ResetMassData();

要得到物体的质量数据，可以通过下面的函数：

float32 GetMass() const;

float32 GetInertia() const;

const b2Vec2& GetLocalCenter() const;

void GetMassData(b2MassData\* data) const;

### 状态信息

物体的有多个方面状态。你可以通过下面的函数高效地访问状态数据：

void SetType(b2BodyType type);

b2BodyType GetType();

void SetBullet(bool flag);

bool IsBullet() const;

void SetSleepingAllowed(bool flag);

bool IsSleepingAllowed() const;

void SetAwake(bool flag);

bool IsAwake() const;

void SetActive(bool flag);

bool IsActive() const;

void SetFixedRotation(bool flag);

bool IsFixedRotation() const;

### 位置和速度

你可以访问一个物体的位置和旋转角，这在你渲染相关游戏角色时很常用。通常情况下，你都是使用Box2D来模拟运动，但你也可以设置位置，尽管并不怎么常用。

bool SetTransform(const b2Vec2& position, float32 angle);

const b2Transform& GetTransform() const;

const b2Vec2& GetPosition() const;

float32 GetAngle() const;

你可以访问本地坐标系及世界坐标下的质心。许多Box2D的内部模拟都使用质心。然而，通常你并不需要访问质心。取而代之，你一般应该关心物体变换。比如，你有个正方形的物体。物体的原点可能在正方形的一个角点，而质心却位于正方形的中心点。

const b2Vec2& GetWorldCenter() const;

const b2Vec2& GetLocalCenter() const;

你可以访问线速度和角速度。线速度是对于质心所言的。所以质量属性改变了，线速度有可能也会改变。

# 夹具

## 关于

回想一下，形状不知道物体的存在，并可独立于物理模拟而被使用。因此Box2D提供b2Fixture类，用于将形状附加到物体上。一个物体可以有零个或多个fixture。拥有多个fixture的物体有时被叫做组合物体。

fixture具有下列属性：

* 关联的形状
* broad-phase代理
* 密度(density)、摩擦(friction)和恢复(restitution)
* 碰撞筛选标记(collision filtering flags)
* 指向父物体的指针
* 用户数据
* 传感器标记(sensor flag)

这些都会在接下来的小节中描述。

## 创建夹具

要创建fixture，先要创始化一个fixture定义，并将定义传到父物体中。

b2FixtureDef fixtureDef;

fixtureDef.shape = &myShape;

fixtureDef.density = 1.0f;

b2Fixture\* myFixture = myBody->CreateFixture(&fixtureDef);

这会创建fixture，并将它附加到物体之上。你不需要保存fixture的指针，因为当它的父物体被摧毁时，fixture也会自动被摧毁。 你可以在单个物体上创建多个fixture。

你可以摧毁父物体上的fixture，来模拟一个可分裂开的物体。你也可以不理会fixture，让物体的析构函数来摧毁附加其上的fixture。

myBody->DestroyFixture(myFixture);

### 密度

fixture的密度用来计算父物体的质量属性。密度值可以为零或者是正数。你所有的fixture都应该使用相似的密度，这样做可以改善堆叠稳定性。

当设置密度的时候，物体的质量不会立即改变。你必须调用ResetMassData ，使之生效。

fixture->SetDensity(5.0f);

body->ResetMassData();

### 摩擦

摩擦可以使对象逼真地沿其它对象滑动。Box2D支持静摩擦和动摩擦，两者都使用相同的参数。摩擦在Box2D中会被精确地模拟，摩擦力的强度与正交力 (称之为库仑摩擦)成正比。摩擦参数通常会设置在0到1之间，但也可是任意的非负数，0意味着没有摩擦，1会产生强摩擦。当计算两个形状之间的摩擦力时，Box2D必须组合两个形状的摩擦参数。这是通过以下公式完成的：

float32 friction;

friction = sqrtf(fixtureA->friction \* fixtureB->friction);

所以当其中一个fixture的摩擦参数为0时，接触的摩擦就为0。

你可以用b2Contact::SetFriction 重载默认的合成摩擦的方法。这通常是在b2ContactListener 的回调中执行的。

### 恢复

恢复可以使对象弹起。恢复的值通常设置在0到1之间。想象一个小球掉落到桌子上，值为0表示着小球不会弹起， 这称为非弹性碰撞。值为1表示小球的速度跟原来一样，只是方向相反，这称为完全弹性碰撞。恢复是通过下面的公式合成的：

float32 restitution;

restitution = b2Max(fixtureA->restitution, fixtureB->restitution);

因为恢复是以这样的方式合成，因此你可以指定一个有弹性的球，而不用指定一个有弹性的地板。Restitution is combined this way so that you can have a bouncy super ball without having a bouncy floor.

你可以用b2Contact:: SetRestitution 重载默认的合成恢复的方法。这通常是在b2ContactListener 的回调中执行的。

当一个形状多次碰撞时，恢复会被近似地模拟，这是因为Box2D使用了迭代求解器。当冲撞速度很小时，Box2D也会使用非弹性碰撞，这是为了防止抖动。参见b2Settings.h中的b2\_velocityThreshold 。

### 筛选

碰撞筛选是为了防止某些fixture之间发生碰撞。比如,你创造了一个骑自行车的角色。你希望自行车与地形之间有碰撞,角色与地形有碰撞,但你不希望角色和自行车之间发生碰撞 (因为它们必须重叠)。Box2D通过种群和分组来支持这样的碰撞筛选。

Box2D支持16个种群。任意fixture你都可以指定它属于哪个种群。你还可以指定这个fixture可以和其它哪些种群发生碰撞。例如,你可以在一个多人游戏中指定玩家之间不会碰撞，怪物之间也不会碰撞，但是玩家和怪物会发生碰撞。这是通过掩码来完成的，例如：

playerFixtureDef.filter.categoryBits = 0x0002;

monsterFixtureDef.filter.categoryBits = 0x0004;

playerFixtureDef.filter.maskBits = 0x0004;

monsterFixtureDef.filter.maskBits = 0x0002;

下面是产生碰撞的规则：

uint16 catA = fixtureA.filter.categoryBits;

uint16 maskA = fixtureA.filter.maskBits;

uint16 catB = fixtureB.filter.categoryBits;

uint16 maskB = fixtureB.filter.maskBits;

if ((catA & maskB) != 0 && (catB & maskA) != 0)

{

// fixtures can collide

}

碰撞分组让你指定一个整数的组索引。你可以让同一个组的所有fixture总是相互碰撞(正索引)或永远不碰撞(负索引)。组索引通常用于一些以某种方式关联的事物，就像自行车的那些部件。在下面的例子中，fixture1 和 fixture2 总是碰撞，而 fixture3 和 fixture4 永远不会碰撞。

fixture1Def.filter.groupIndex = 2;

fixture2Def.filter.groupIndex = 2;

fixture3Def.filter.groupIndex = -8;

fixture4Def.filter.groupIndex = -8;

如果组索引不同，碰撞筛选就会按照种群和掩码来进行。换句话说，分组筛选与种群筛选相比，具有更高的优选级。

注意在 Box2D 中还有其它的碰撞筛选，这里是一个列表：

* static上的fixture只会与dynamic物体上的fixture发生碰撞。
* kinematic物体 只会和dynamic物体碰撞。
* 同一个物体上的fixture永远不会相互碰撞。
* 如果两个物体用关节连接起来，物体上面的fixture可以选择启用或禁止它们之间相互碰撞。

有时你可能希望在形状创建之后去改变其碰撞筛选。 你可以使用b2Shape::GetFilterData 和b2Shape::SetFilterData来访问和设置已存在的fixture的b2FilterData结构。注意就算修改了筛选数据，到下一个时间步 为止，现在的接触并不会被增加或删除（参见world类）。

## 传感器

有时候游戏逻辑需要判断两个fixture是否相交，而不想有碰撞反应。这可以通过传感器(sensor)来完成。传感器也是个fixture，但它只会侦测碰撞，而不产生其它反应。

你可以将任意fixture标记为传感器。传感器可以是static、kinematic或dynamic的。记住，每个物体上可以有多个fixture，传感器和实体fixture是可以混合存在的。而且，只有至少一个物体是dynamic的，传感器才会产生接触事件，而kinematic 与kinematic 、kinematic 与static ，或者static 与static 之间都不会产生接触事件。

传感器不会生成接触点。这里有两种方法得到传感器的状态：

1. b2Contact::IsTouching
2. b2ContactListener::BeginContact 和 EndContact

# 关节

## 关于

关节用于把物体约束到世界，或约束到其它物体上。在游戏中，典型例子有木偶，跷跷板和滑轮。用不同的方式将关节结合起来使用，可以创造出有趣的运动。

有些关节提供了限制(limit)，使你可以控制运动的范围。有些关节还提供了马达(motor)，它可以以指定的速度驱动关节一直运动，直到你指定了更大的力或扭矩来抵消这种运动。

关节马达有许多不同的用途。你可以使用关节来控制位置，只要提供一个与目标之距离成正比例的关节速度即可。你还可以模拟关节摩擦：将关节速度置零，并且提供一个小的、但有效的最大力或扭矩；那么马达就会努力保持关节不动，直到负载变得过大为止。

## 关节定义

每种关节类型都有各自的定义(definition)，它们都派生自b2JointDef。所有的关节都连接两个不同的物体，其中一个物体有可能是静态的。关节也可以连接两个static或者kinematic类型的物体，但这没有任何实际用途，只会浪费处理器时间。

你可以为任何一种关节类型指定用户数据。你还可以提供一个标记，用于防止用关节相连的物体之间发生碰撞。实际上, 这是默认行为。你也可以通过设置 collideConnected，来允许相连的物体之间发生碰撞。

很多关节定义需要你提供一些几何数据。一个关节常常需要一个锚点(anchor point)来定义，这是固定于相接物体中的点。Box2D要求这些点在局部坐标系中指定，这样，即便当前物体的变化违反了关节约束(joint constraint)，关节还是可以被指定——这通常会发生在游戏保存或载入进度时。另外，有些关节定义需要知道物体之间默认的相对角度。这样才能正确地约束旋转。

初始化几何数据可能有些乏味。所以很多关节提供了初始化函数，使用当前的物体的形状，来消除大部分工作。然而，这些初始化函数通常只应用于原型，在产品代码中应该直接地定义几何数据。这能使关节行为更具健壮性。

其余的关节定义数据依赖于关节的类型。下面我们来介绍它们。The rest of the joint definition data depends on the joint type. We cover these now.

## Joint Factory

关节使用world的工厂方法来创建和摧毁。这引入一个老问题：Joints are created and destroyed using the world factory methods. This brings up an old issue:

**Caution**

不要使用new或malloc在栈(stack)或堆(heap)中创建关节。你想创建或摧毁物体和关节，必须使用b2World类中对应的创建或摧毁函数。Don't try to create a joint on the stack or on the heap using new or malloc. You must create and destroy bodies and joints using the create and destroy methods of the b2World class.

这里有个例子，展示了旋转关节(revolute joint)从创建到摧毁的过程:Here's an example of the lifetime of a revolute joint:

b2RevoluteJointDef jointDef;

jointDef.bodyA = myBodyA;

jointDef.bodyB = myBodyB;

jointDef.anchorPoint = myBodyA->GetCenterPosition();

b2RevoluteJoint\* joint = (b2RevoluteJoint\*)myWorld->CreateJoint(&jointDef);

... do stuff ...

myWorld->DestroyJoint(joint);

joint = NULL;

一个很好的习惯: 当对象摧毁后，就将对应的指针清零。不清零的话，当你试图再次使用这个指针，程序在特定时候会崩溃。It is always good to nullify your pointer after they are destroyed. This will make the program crash in a controlled manner if you try to reuse the pointer.

关节的生命周期并不简单，要特别留心下面的警告，敲响警钟。The lifetime of a joint is not simple. Heed this warning well:

**Caution**

Joints are destroyed when an attached body is destroyed.

This precaution is not always necessary. You may organize your game engine so that joints are always destroyed before the attached bodies. In this case you don't need to implement the listener class. See the section on Implicit Destruction for details.

## Using Joints

Many simulations create the joints and don't access them again until they are destroyed. However, there is a lot of useful data contained in joints that you can use to create a rich simulation.

First of all, you can get the bodies, anchor points, and user data from a joint.

b2Body\* GetBodyA();

b2Body\* GetBodyB();

b2Vec2 GetAnchorA();

b2Vec2 GetAnchorB();

void\* GetUserData();

All joints have a reaction force and torque. This the reaction force applied to body 2 at the anchor point. You can use reaction forces to break joints or trigger other game events. These functions may do some computations, so don't call them if you don't need the result.

b2Vec2 GetReactionForce();

float32 GetReactionTorque();

## Distance Joint

One of the simplest joint is a distance joint which says that the distance between two points on two bodies must be constant. When you specify a distance joint the two bodies should already be in place. Then you specify the two anchor points in world coordinates. The first anchor point is connected to body 1, and the second anchor point is connected to body 2. These points imply the length of the distance constraint.



Here is an example of a distance joint definition. In this case we decide to allow the bodies to collide.

b2DistanceJointDef jointDef;

jointDef.Initialize(myBodyA, myBodyB, worldAnchorOnBodyA, worldAnchorOnBodyB);

jointDef.collideConnected = true;

The distance joint can also be made soft, like a spring-damper connection. See the Web example in the testbed to see how this behaves.

Softness is achieved by tuning two constants in the definition: frequency and damping ratio. Think of the frequency as the frequency of a harmonic oscillator (like a guitar string). The frequency is specified in Hertz. Typically the frequency should be less than a half the frequency of the time step. So if you are using a 60Hz time step, the frequency of the distance joint should be less than 30Hz. The reason is related to the Nyquist frequency.

The damping ratio is non-dimensional and is typically between 0 and 1, but can be larger. At 1, the damping is critical (all oscillations should vanish).

jointDef.frequencyHz = 4.0f;

jointDef.dampingRatio = 0.5f;

## Revolute Joint

A revolute joint forces two bodies to share a common anchor point, often called a hinge point. The revolute joint has a single degree of freedom: the relative rotation of the two bodies. This is called the joint angle.



To specify a revolute you need to provide two bodies and a single anchor point in world space. The initialization function assumes that the bodies are already in the correct position.

In this example, two bodies are connected by a revolute joint at the first body's center of mass.

b2RevoluteJointDef jointDef;

jointDef.Initialize(myBodyA, myBodyB, myBodyA->GetWorldCenter());

The revolute joint angle is positive when bodyB rotates CCW about the angle point. Like all angles in Box2D, the revolute angle is measured in radians. By convention the revolute joint angle is zero when the joint is created using Initialize(), regardless of the current rotation of the two bodies.

In some cases you might wish to control the joint angle. For this, the revolute joint can optionally simulate a joint limit and/or a motor.

A joint limit forces the joint angle to remain between a lower and upper bound. The limit will apply as much torque as needed to make this happen. The limit range should include zero, otherwise the joint will lurch when the simulation begins.

A joint motor allows you to specify the joint speed (the time derivative of the angle). The speed can be negative or positive. A motor can have infinite force, but this is usually not desirable. Recall the eternal question:

"What happens when an irresistible force meets an immovable object?"

I can tell you it's not pretty. So you can provide a maximum torque for the joint motor. The joint motor will maintain the specified speed unless the required torque exceeds the specified maximum. When the maximum torque is exceeded, the joint will slow down and can even reverse.

You can use a joint motor to simulate joint friction. Just set the joint speed to zero, and set the maximum torque to some small, but significant value. The motor will try to prevent the joint from rotating, but will yield to a significant load.

Here's a revision of the revolute joint definition above; this time the joint has a limit and a motor enabled. The motor is setup to simulate joint friction.

b2RevoluteJointDef jointDef;

jointDef.Initialize(bodyA, bodyB, myBodyA->GetWorldCenter());

jointDef.lowerAngle = -0.5f \* b2\_pi; // -90 degrees

jointDef.upperAngle = 0.25f \* b2\_pi; // 45 degrees

jointDef.enableLimit = true;

jointDef.maxMotorTorque = 10.0f;

jointDef.motorSpeed = 0.0f;

jointDef.enableMotor = true;

You can access a revolute joint's angle, speed, and motor torque.

float32 GetJointAngle() const;

float32 GetJointSpeed() const;

float32 GetMotorTorque() const;

You also update the motor parameters each step.

void SetMotorSpeed(float32 speed);

void SetMaxMotorTorque(float32 torque);

Joint motors have some interesting abilities. You can update the joint speed every time step so you can make the joint move back-and-forth like a sine-wave or according to whatever function you want.

... Game Loop Begin ...

myJoint->SetMotorSpeed(cosf(0.5f \* time));

... Game Loop End ...

You can also use joint motors to track a desired joint angle. For example:

... Game Loop Begin ...

float32 angleError = myJoint->GetJointAngle() - angleTarget;

float32 gain = 0.1f;

myJoint->SetMotorSpeed(-gain \* angleError);

... Game Loop End ...

Generally your gain parameter should not be too large. Otherwise your joint may become unstable.

## Prismatic Joint

A prismatic joint allows for relative translation of two bodies along a specified axis. A prismatic joint prevents relative rotation. Therefore, a prismatic joint has a single degree of freedom.



The prismatic joint definition is similar to the revolute joint description; just substitute translation for angle and force for torque. Using this analogy provides an example prismatic joint definition with a joint limit and a friction motor:

b2PrismaticJointDef jointDef;

b2Vec2 worldAxis(1.0f, 0.0f);

jointDef.Initialize(myBodyA, myBodyB, myBodyA->GetWorldCenter(), worldAxis);

jointDef.lowerTranslation = -5.0f;

jointDef.upperTranslation = 2.5f;

jointDef.enableLimit = true;

jointDef.maxMotorForce = 1.0f;

jointDef.motorSpeed = 0.0f;

jointDef.enableMotor = true;

The revolute joint has an implicit axis coming out of the screen. The prismatic joint needs an explicit axis parallel to the screen. This axis is fixed in the two bodies and follows their motion.

Like the revolute joint, the prismatic joint translation is zero when the joint is created using Initialize(). So be sure zero is between your lower and upper translation limits.

Using a prismatic joint is similar to using a revolute joint. Here are the relevant member functions:

float32 GetJointTranslation() const;

float32 GetJointSpeed() const;

float32 GetMotorForce() const;

void SetMotorSpeed(float32 speed);

void SetMotorForce(float32 force);

## Pulley Joint

A pulley is used to create an idealized pulley. The pulley connects two bodies to ground and to each other. As one body goes up, the other goes down. The total length of the pulley rope is conserved according to the initial configuration.

length1 + length2 == constant

You can supply a ratio that simulates a block and tackle. This causes one side of the pulley to extend faster than the other. At the same time the constraint force is smaller on one side than the other. You can use this to create mechanical leverage.

length1 + ratio \* length2 == constant

For example, if the ratio is 2, then length1 will vary at twice the rate of length2. Also the force in the rope attached to body1 will have half the constraint force as the rope attached to body2.



Pulleys can be troublesome when one side is fully extended. The rope on the other side will have zero length. At this point the constraint equations become singular (bad). You should configure collision shapes to prevent this.

Here is an example pulley definition:

b2Vec2 anchor1 = myBody1->GetWorldCenter();

b2Vec2 anchor2 = myBody2->GetWorldCenter();

b2Vec2 groundAnchor1(p1.x, p1.y + 10.0f);

b2Vec2 groundAnchor2(p2.x, p2.y + 12.0f);

float32 ratio = 1.0f;

b2PulleyJointDef jointDef;

jointDef.Initialize(myBody1, myBody2, groundAnchor1, groundAnchor2, anchor1, anchor2, ratio);

Pulley joints provide the current lengths.

float32 GetLengthA() const;

float32 GetLengthB() const;

## Gear Joint

If you want to create a sophisticated mechanical contraption you might want to use gears. In principle you can create gears in Box2D by using compound shapes to model gear teeth. This is not very efficient and might be tedious to author. You also have to be careful to line up the gears so the teeth mesh smoothly. Box2D has a simpler method of creating gears: the gear joint.



The gear joint can only connect revolute and/or prismatic joints.

Like the pulley ratio, you can specify a gear ratio. However, in this case the gear ratio can be negative. Also keep in mind that when one joint is a revolute joint (angular) and the other joint is prismatic (translation), and then the gear ratio will have units of length or one over length.

coordinate1 + ratio \* coordinate2 == constant

Here is an example gear joint. The bodies myBodyA and myBodyB are any bodies from the two joints, as long as they are not the same bodies.

b2GearJointDef jointDef;

jointDef.bodyA = myBodyA;

jointDef.bodyB = myBodyB;

jointDef.joint1 = myRevoluteJoint;

jointDef.joint2 = myPrismaticJoint;

jointDef.ratio = 2.0f \* b2\_pi / myLength;

Note that the gear joint depends on two other joints. This creates a fragile situation. What happens if those joints are deleted?

**Caution**

Always delete gear joints before the revolute/prismatic joints on the gears. Otherwise your code will crash in a bad way due to the orphaned joint pointers in the gear joint. You should also delete the gear joint before you delete any of the bodies involved.

## Mouse Joint

The mouse joint is used in the testbed to manipulate bodies with the mouse. It attempts to drive a point on a body towards the current position of the cursor. There is no restriction on rotation.

The mouse joint definition has a target point, maximum force, frequency, and damping ratio. The target point initially coincides with the body’s anchor point. The maximum force is used to prevent violent reactions when multiple dynamic bodies interact. You can make this as large as you like. The frequency and damping ratio are used to create a spring/damper effect similar to the distance joint.

Many users have tried to adapt the mouse joint for game play. Users often want to achieve precise positioning and instantaneous response. The mouse joint doesn’t work very well in that context. You may wish to consider using kinematic bodies instead.

## Wheel Joint

The wheel joint restricts a point on bodyB to a line on bodyA. The wheel joint also provides a suspension spring. See b2WheelJoint.h and Car.h for details.



## Weld Joint

The weld joint attempts to constrain all relative motion between two bodies. See the Cantilever.h in the testbed to see how the weld joint behaves.

It is tempting to use the weld joint to define breakable structures. However, the Box2D solver is iterative so the joints are a bit soft. So chains of bodies connected by weld joints will flex.

Instead it is better to create breakable bodies starting with a single body with multiple fixtures. When the body breaks, you can destroy a fixture and recreate it on a new body. See the Breakable example in the testbed.

## Rope Joint

The rope joint restricts the maximum distance between two points. This can be useful to prevent chains of bodies from stretching, even under high load. See b2RopeJoint.h and RopeJoint.h for details.

## Friction Joint

The friction joint is used for top-down friction. The joint provides 2D translational friction and angular friction. See b2FrictionJoint.h and ApplyForce.h for details.

## Motor Joint

A motor joint lets you control the motion of a body by specifying target position and rotation offsets. You can set the maximum motor force and torque that will be applied to reach the target position and rotation. If the body is blocked, it will stop and the contact forces will be proportional the maximum motor force and torque. See b2MotorJoint and MotorJoint.h for details.

# Contacts

## About

Contacts are objects created by Box2D to manage collision between two fixtures. If the fixture has children, such as a chain shape, then a contact exists for each relevant child. There are different kinds of contacts, derived from b2Contact, for managing contact between different kinds of fixtures. For example there is a contact class for managing polygon-polygon collision and another contact class for managing circle-circle collision.

Here is some terminology associated with contacts.

### contact point

A contact point is a point where two shapes touch. Box2D approximates contact with a small number of points.

### contact normal

A contact normal is a unit vector that points from one shape to another. By convention, the normal points from fixtureA to fixtureB.

### contact separation

Separation is the opposite of penetration. Separation is negative when shapes overlap. It is possible that future versions of Box2D will create contact points with positive separation, so you may want to check the sign when contact points are reported.

### contact manifold

Contact between two convex polygons may generate up to 2 contact points. Both of these points use the same normal, so they are grouped into a contact manifold, which is an approximation of a continuous region of contact.

### normal impulse

The normal force is the force applied at a contact point to prevent the shapes from penetrating. For convenience, Box2D works with impulses. The normal impulse is just the normal force multiplied by the time step.

### tangent impulse

The tangent force is generated at a contact point to simulate friction. For convenience, this is stored as an impulse.

### contact ids

Box2D tries to re-use the contact force results from a time step as the initial guess for the next time step. Box2D uses contact ids to match contact points across time steps. The ids contain geometric features indices that help to distinguish one contact point from another.

Contacts are created when two fixture’s AABBs overlap. Sometimes collision filtering will prevent the creation of contacts. Contacts are destroyed with the AABBs cease to overlap.

So you might gather that there may be contacts created for fixtures that are not touching (just their AABBs). Well, this is correct. It's a "chicken or egg" problem. We don't know if we need a contact object until one is created to analyze the collision. We could delete the contact right away if the shapes are not touching, or we can just wait until the AABBs stop overlapping. Box2D takes the latter approach because it lets the system cache information to improve performance.

## Contact Class

As mentioned before, the contact class is created and destroyed by Box2D. Contact objects are not created by the user. However, you are able to access the contact class and interact with it.

You can access the raw contact manifold:

b2Manifold\* GetManifold();

const b2Manifold\* GetManifold() const;

You can potentially modify the manifold, but this is generally not supported and is for advanced usage.

There is a helper function to get the b2WorldManifold:

void GetWorldManifold(b2WorldManifold\* worldManifold) const;

This uses the current positions of the bodies to compute world positions of the contact points.

Sensors do not create manifolds, so for them use:

bool touching = sensorContact->IsTouching();

This function also works for non-sensors.

You can get the fixtures from a contact. From those you can get the bodies.

b2Fixture\* fixtureA = myContact->GetFixtureA();

b2Body\* bodyA = fixtureA->GetBody();

MyActor\* actorA = (MyActor\*)bodyA->GetUserData();

You can disable a contact. This only works inside the b2ContactListener::PreSolve event, discussed below.

## Accessing Contacts

You can get access to contacts in several ways. You can access the contacts directly on the world and body structures. You can also implement a contact listener.

You can iterate over all contacts in the world:

for (b2Contact\* c = myWorld->GetContactList(); c; c = c->GetNext())

{

// process c

}

You can also iterate over all the contacts on a body. These are stored in a graph using a contact edge structure.

for (b2ContactEdge\* ce = myBody->GetContactList(); ce; ce = ce->next)

{

b2Contact\* c = ce->contact;

// process c

}

You can also access contacts using the contact listener that is described below.

**Caution**

Accessing contacts off b2World and b2Body may miss some transient contacts that occur in the middle of the time step. Use b2ContactListener to get the most accurate results.

## Contact Listener

You can receive contact data by implementing b2ContactListener. The contact listener supports several events: begin, end, pre-solve, and post-solve.

class MyContactListener : public b2ContactListener

{

public:

void BeginContact(b2Contact\* contact)

{ /\* handle begin event \*/ }

void EndContact(b2Contact\* contact)

{ /\* handle end event \*/ }

void PreSolve(b2Contact\* contact, const b2Manifold\* oldManifold)

{ /\* handle pre-solve event \*/ }

void PostSolve(b2Contact\* contact, const b2ContactImpulse\* impulse)

{ /\* handle post-solve event \*/ }

};

**Caution**

Do not keep a reference to the pointers sent to b2ContactListener. Instead make a deep copy of the contact point data into your own buffer. The example below shows one way of doing this.

At run-time you can create an instance of the listener and register it with b2World::SetContactListener. Be sure your listener remains in scope while the world object exists.

### Begin Contact Event

This is called when two fixtures begin to overlap. This is called for sensors and non-sensors. This event can only occur inside the time step.

### End Contact Event

This is called when two fixtures cease to overlap. This is called for sensors and non-sensors. This may be called when a body is destroyed, so this event can occur outside the time step.

### Pre-Solve Event

This is called after collision detection, but before collision resolution. This gives you a chance to disable the contact based on the current configuration. For example, you can implement a one-sided platform using this callback and calling b2Contact::SetEnabled(false). The contact will be re-enabled each time through collision processing, so you will need to disable the contact every time-step. The pre-solve event may be fired multiple times per time step per contact due to continuous collision detection.

void PreSolve(b2Contact\* contact, const b2Manifold\* oldManifold)

{

b2WorldManifold worldManifold;

contact->GetWorldManifold(&worldManifold);

if (worldManifold.normal.y < -0.5f)

{

contact->SetEnabled(false);

}

}

The pre-solve event is also a good place to determine the point state and the approach velocity of collisions.

void PreSolve(b2Contact\* contact, const b2Manifold\* oldManifold)

{

b2WorldManifold worldManifold;

contact->GetWorldManifold(&worldManifold);

b2PointState state1[2], state2[2];

b2GetPointStates(state1, state2, oldManifold, contact->GetManifold());

if (state2[0] == b2\_addState)

{

const b2Body\* bodyA = contact->GetFixtureA()->GetBody();

const b2Body\* bodyB = contact->GetFixtureB()->GetBody();

b2Vec2 point = worldManifold.points[0];

b2Vec2 vA = bodyA->GetLinearVelocityFromWorldPoint(point);

b2Vec2 vB = bodyB->GetLinearVelocityFromWorldPoint(point);

float32 approachVelocity = b2Dot(vB – vA, worldManifold.normal);

if (approachVelocity > 1.0f)

{

MyPlayCollisionSound();

}

}

}

### Post-Solve Event

The post solve event is where you can gather collision impulse results. If you don’t care about the impulses, you should probably just implement the pre-solve event.

It is tempting to implement game logic that alters the physics world inside a contact callback. For example, you may have a collision that applies damage and try to destroy the associated actor and its rigid body. However, Box2D does not allow you to alter the physics world inside a callback because you might destroy objects that Box2D is currently processing, leading to orphaned pointers.

The recommended practice for processing contact points is to buffer all contact data that you care about and process it after the time step. You should always process the contact points immediately after the time step; otherwise some other client code might alter the physics world, invalidating the contact buffer. When you process the contact buffer you can alter the physics world, but you still need to be careful that you don't orphan pointers stored in the contact point buffer. The testbed has example contact point processing that is safe from orphaned pointers.

This code from the CollisionProcessing test shows how to handle orphaned bodies when processing the contact buffer. Here is an excerpt. Be sure to read the comments in the listing. This code assumes that all contact points have been buffered in the b2ContactPoint array m\_points.

// We are going to destroy some bodies according to contact

// points. We must buffer the bodies that should be destroyed

// because they may belong to multiple contact points.

const int32 k\_maxNuke = 6;

b2Body\* nuke[k\_maxNuke];

int32 nukeCount = 0;

// Traverse the contact buffer. Destroy bodies that

// are touching heavier bodies.

for (int32 i = 0; i < m\_pointCount; ++i)

{

ContactPoint\* point = m\_points + i;

b2Body\* bodyA = point->fixtureA->GetBody();

b2Body\* bodyB = point->FixtureB->GetBody();

float32 massA = bodyA->GetMass();

float32 massB = bodyB->GetMass();

if (massA > 0.0f && massB > 0.0f)

{

if (massB > massA)

{

nuke[nukeCount++] = bodyA;

}

else

{

nuke[nukeCount++] = bodyB;

}

if (nukeCount == k\_maxNuke)

{

break;

}

}

}

// Sort the nuke array to group duplicates.

std::sort(nuke, nuke + nukeCount);

// Destroy the bodies, skipping duplicates.

int32 i = 0;

while (i < nukeCount)

{

b2Body\* b = nuke[i++];

while (i < nukeCount && nuke[i] == b)

{

++i;

}

m\_world->DestroyBody(b);

}

## Contact Filtering

Often in a game you don't want all objects to collide. For example, you may want to create a door that only certain characters can pass through. This is called contact filtering, because some interactions are filtered out.

Box2D allows you to achieve custom contact filtering by implementing a b2ContactFilter class. This class requires you to implement a ShouldCollide function that receives two b2Shape pointers. Your function returns true if the shapes should collide.

The default implementation of ShouldCollide uses the b2FilterData defined in Chapter 6, Fixtures.

bool b2ContactFilter::ShouldCollide(b2Fixture\* fixtureA, b2Fixture\* fixtureB)

{

const b2Filter& filterA = fixtureA->GetFilterData();

const b2Filter& filterB = fixtureB->GetFilterData();

if (filterA.groupIndex == filterB.groupIndex && filterA.groupIndex != 0)

{

return filterA.groupIndex > 0;

}

bool collide = (filterA.maskBits & filterB.categoryBits) != 0 &&

(filterA.categoryBits & filterB.maskBits) != 0;

return collide;

}

At run-time you can create an instance of your contact filter and register it with b2World::SetContactFilter. Make sure your filter stays in scope while the world exists.

MyContactFilter filter;

world->SetContactFilter(&filter);

// filter remains in scope …

# World Class

### About

The b2World class contains the bodies and joints. It manages all aspects of the simulation and allows for asynchronous queries (like AABB queries and ray-casts). Much of your interactions with Box2D will be with a b2World object.

### Creating and Destroying a World

Creating a world is fairly simple. You just need to provide a gravity vector and a Boolean indicating if bodies can sleep. Usually you will create and destroy a world using new and delete.

b2World\* myWorld = new b2World(gravity, doSleep);

... do stuff ...

delete myWorld;

### Using a World

The world class contains factories for creating and destroying bodies and joints. These factories are discussed later in the sections on bodies and joints. There are some other interactions with b2World that I will cover now.

### Simulation

The world class is used to drive the simulation. You specify a time step and a velocity and position iteration count. For example:

float32 timeStep = 1.0f / 60.f;

int32 velocityIterations = 10;

int32 positionIterations = 8;

myWorld->Step(timeStep, velocityIterations, positionIterations);

After the time step you can examine your bodies and joints for information. Most likely you will grab the position off the bodies so that you can update your actors and render them. You can perform the time step anywhere in your game loop, but you should be aware of the order of things. For example, you must create bodies before the time step if you want to get collision results for the new bodies in that frame.

As I discussed above in the HelloWorld tutorial, you should use a fixed time step. By using a larger time step you can improve performance in low frame rate scenarios. But generally you should use a time step no larger than 1/30 seconds. A time step of 1/60 seconds will usually deliver a high quality simulation.

The iteration count controls how many times the constraint solver sweeps over all the contacts and joints in the world. More iteration always yields a better simulation. But don't trade a small time step for a large iteration count. 60Hz and 10 iterations is far better than 30Hz and 20 iterations.

After stepping, you should clear any forces you have applied to your bodies. This is done with the command b2World::ClearForces. This lets you take multiple sub-steps with the same force field.

myWorld->ClearForces();

### Exploring the World

The world is a container for bodies, contacts, and joints. You can grab the body, contact, and joint lists off the world and iterate over them. For example, this code wakes up all the bodies in the world:

for (b2Body\* b = myWorld->GetBodyList(); b; b = b->GetNext())

{

b->SetAwake(true);

}

Unfortunately real programs can be more complicated. For example, the following code is broken:

for (b2Body\* b = myWorld->GetBodyList(); b; b = b->GetNext())

{

GameActor\* myActor = (GameActor\*)b->GetUserData();

if (myActor->IsDead())

{

myWorld->DestroyBody(b); // ERROR: now GetNext returns garbage.

}

}

Everything goes ok until a body is destroyed. Once a body is destroyed, its next pointer becomes invalid. So the call to b2Body::GetNext() will return garbage. The solution to this is to copy the next pointer before destroying the body.

b2Body\* node = myWorld->GetBodyList();

while (node)

{

b2Body\* b = node;

node = node->GetNext();

GameActor\* myActor = (GameActor\*)b->GetUserData();

if (myActor->IsDead())

{

myWorld->DestroyBody(b);

}

}

This safely destroys the current body. However, you may want to call a game function that may destroy multiple bodies. In this case you need to be very careful. The solution is application specific, but for convenience I'll show one method of solving the problem.

b2Body\* node = myWorld->GetBodyList();

while (node)

{

b2Body\* b = node;

node = node->GetNext();

GameActor\* myActor = (GameActor\*)b->GetUserData();

if (myActor->IsDead())

{

bool otherBodiesDestroyed = GameCrazyBodyDestroyer(b);

if (otherBodiesDestroyed)

{

node = myWorld->GetBodyList();

}

}

}

Obviously to make this work, GameCrazyBodyDestroyer must be honest about what it has destroyed.

### AABB Queries

Sometimes you want to determine all the shapes in a region. The b2World class has a fast log(N) method for this using the broad-phase data structure. You provide an AABB in world coordinates and an implementation of b2QueryCallback. The world calls your class with each fixture whose AABB overlaps the query AABB. Return true to continue the query, otherwise return false. For example, the following code finds all the fixtures that potentially intersect a specified AABB and wakes up all of the associated bodies.

class MyQueryCallback : public b2QueryCallback

{

public:

bool ReportFixture(b2Fixture\* fixture)

{

b2Body\* body = fixture->GetBody();

body->SetAwake(true);

// Return true to continue the query.

return true;

}

};

...

MyQueryCallback callback;

b2AABB aabb;

aabb.lowerBound.Set(-1.0f, -1.0f);

aabb.upperBound.Set(1.0f, 1.0f);

myWorld->Query(&callback, aabb);

You cannot make any assumptions about the order of the callbacks.

### Ray Casts

You can use ray casts to do line-of-sight checks, fire guns, etc. You perform a ray cast by implementing a callback class and providing the start and end points. The world class calls your class with each fixture hit by the ray. Your callback is provided with the fixture, the point of intersection, the unit normal vector, and the fractional distance along the ray. You cannot make any assumptions about the order of the callbacks.

You control the continuation of the ray cast by returning a fraction. Returning a fraction of zero indicates the ray cast should be terminated. A fraction of one indicates the ray cast should continue as if no hit occurred. If you return the fraction from the argument list, the ray will be clipped to the current intersection point. So you can ray cast any shape, ray cast all shapes, or ray cast the closest shape by returning the appropriate fraction.

You may also return of fraction of -1 to filter the fixture. Then the ray cast will proceed as if the fixture does not exist.

Here is an example:

// This class captures the closest hit shape.

class MyRayCastCallback : public b2RayCastCallback

{

public:

MyRayCastCallback()

{

m\_fixture = NULL;

}

float32 ReportFixture(b2Fixture\* fixture, const b2Vec2& point,

const b2Vec2& normal, float32 fraction)

{

m\_fixture = fixture;

m\_point = point;

m\_normal = normal;

m\_fraction = fraction;

return fraction;

}

b2Fixture\* m\_fixture;

b2Vec2 m\_point;

b2Vec2 m\_normal;

float32 m\_fraction;

};

MyRayCastCallback callback;

b2Vec2 point1(-1.0f, 0.0f);

b2Vec2 point2(3.0f, 1.0f);

myWorld->RayCast(&callback, point1, point2);

**Caution**

Due to round-off errors, ray casts can sneak through small cracks between polygons in your static environment. If this is not acceptable in your application, please enlarge your polygons slightly.

void SetLinearVelocity(const b2Vec2& v);

b2Vec2 GetLinearVelocity() const;

void SetAngularVelocity(float32 omega);

float32 GetAngularVelocity() const;

### Forces and Impulses

You can apply forces, torques, and impulses to a body. When you apply a force or an impulse, you provide a world point where the load is applied. This often results in a torque about the center of mass.

void ApplyForce(const b2Vec2& force, const b2Vec2& point);

void ApplyTorque(float32 torque);

void ApplyLinearImpulse(const b2Vec2& impulse, const b2Vec2& point);

void ApplyAngularImpulse(float32 impulse);

Applying a force, torque, or impulse wakes the body. Sometimes this is undesirable. For example, you may be applying a steady force and want to allow the body to sleep to improve performance. In this case you can use the following code.

if (myBody->IsAwake() == true)

{

myBody->ApplyForce(myForce, myPoint);

}

### Coordinate Transformations

The body class has some utility functions to help you transform points and vectors between local and world space. If you don't understand these concepts, please read "Essential Mathematics for Games and Interactive Applications" by Jim Van Verth and Lars Bishop. These functions are efficient (when inlined).

b2Vec2 GetWorldPoint(const b2Vec2& localPoint);

b2Vec2 GetWorldVector(const b2Vec2& localVector);

b2Vec2 GetLocalPoint(const b2Vec2& worldPoint);

b2Vec2 GetLocalVector(const b2Vec2& worldVector);

### Lists

You can iterate over a body's fixtures. This is mainly useful if you need to access the fixture's user data.

for (b2Fixture\* f = body->GetFixtureList(); f; f = f->GetNext())

{

MyFixtureData\* data = (MyFixtureData\*)f->GetUserData();

... do something with data ...

}

You can similarly iterate over the body's joint list.

The body also provides a list of associated contacts. You can use this to get information about the current contacts. Be careful, because the contact list may not contain all the contacts that existed during the previous time step.

# Loose Ends

## 用户数据

b2Fixture, b2Body 和 b2Joint 类都允许你通过一个 void 指针来附加用户数据。当你测试Box2D数据结构，并使其跟自己游戏引擎中的对象结合起来时，这样做是比较方便的。

举个典型的例子，角色上附有物体，并在物体中附加角色的指针，这就构成了一个循环引用。如果你有角色(actor)，你就能得到物体。如果你有物体，你也能得到角色。

GameActor\* actor = GameCreateActor();

b2BodyDef bodyDef;

bodyDef.userData = actor;

actor->body = box2Dworld->CreateBody(&bodyDef);

一些需要用户数据的例子：

* 使用碰撞结果给角色施加伤害效果。
* 当玩家进入一个包围盒(axis-aligned box)时，触发脚本事件。
* 当Box2D通知你关节将要被摧毁时，去访问某个游戏结构。

记住，用户数据是可选的，并且能放入任何东西。然而，你需要确保一致性。例如，如果你想在body中保存actor的指针，那你就应该在所有的 body中都保存actor指针。不要在一个body中保存actor指针，却在另一个body中保存foo指针。将一个actor指针强制转成foo指 针，可能会导致程序崩溃。

用户数据指针默认为NULL。

对于fixture来说，你可以定义一个用户数据结构来存储游戏特定的信息。例如材料类型、特效钩子、音效钩子，等等。

struct FixtureUserData

{

int materialIndex;

…

};

FixtureUserData myData = new FixtureUserData;

myData->materialIndex = 2;

b2FixtureDef fixtureDef;

fixtureDef.shape = &someShape;

fixtureDef.userData = myData;

b2Fixture\* fixture = body->CreateFixture(&fixtureDef);

…

delete fixture->GetUserData();

fixture->SetUserData(NULL);

body->DestroyFixture(fixture);

## 

## Implicit Destruction

Box2D doesn't use reference counting. So if you destroy a body it is really gone. Accessing a pointer to a destroyed body has undefined behavior. In other words, your program will likely crash and burn. To help fix these problems, the debug build memory manager fills destroyed entities with FDFDFDFD. This can help find problems more easily in some cases.

If you destroy a Box2D entity, it is up to you to make sure you remove all references to the destroyed object. This is easy if you only have a single reference to the entity. If you have multiple references, you might consider implementing a handle class to wrap the raw pointer.

Often when using Box2D you will create and destroy many bodies, shapes, and joints. Managing these entities is somewhat automated by Box2D. If you destroy a body then all associated shapes and joints are automatically destroyed. This is called implicit destruction.

When you destroy a body, all its attached shapes, joints, and contacts are destroyed. This is called implicit destruction. Any body connected to one of those joints and/or contacts is woken. This process is usually convenient. However, you must be aware of one crucial issue:

**Caution**

When a body is destroyed, all fixtures and joints attached to the body are automatically destroyed. You must nullify any pointers you have to those shapes and joints. Otherwise, your program will die horribly if you try to access or destroy those shapes or joints later.

To help you nullify your joint pointers, Box2D provides a listener class named b2DestructionListener that you can implement and provide to your world object. Then the world object will notify you when a joint is going to be implicitly destroyed

Note that there no notification when a joint or fixture is explicitly destroyed. In this case ownership is clear and you can perform the necessary cleanup on the spot. If you like, you can call your own implementation of b2DestructionListener to keep cleanup code centralized.

Implicit destruction is a great convenience in many cases. It can also make your program fall apart. You may store pointers to shapes and joints somewhere in your code. These pointers become orphaned when an associated body is destroyed. The situation becomes worse when you consider that joints are often created by a part of the code unrelated to management of the associated body. For example, the testbed creates a b2MouseJoint for interactive manipulation of bodies on the screen.

Box2D provides a callback mechanism to inform your application when implicit destruction occurs. This gives your application a chance to nullify the orphaned pointers. This callback mechanism is described later in this manual.

You can implement a b2DestructionListener that allows b2World to inform you when a shape or joint is implicitly destroyed because an associated body was destroyed. This will help prevent your code from accessing orphaned pointers.

class MyDestructionListener : public b2DestructionListener

{

void SayGoodbye(b2Joint\* joint)

{

// remove all references to joint.

}

};

You can then register an instance of your destruction listener with your world object. You should do this during world initialization.

myWorld->SetListener(myDestructionListener);

## Pixels and Coordinate Systems

Recall that Box2D uses MKS (meters, kilograms, and seconds) units and radians for angles. You may have trouble working with meters because your game is expressed in terms of pixels. To deal with this in the testbed I have the whole *game* work in meters and just use an OpenGL viewport transformation to scale the world into screen space.

float lowerX = -25.0f, upperX = 25.0f, lowerY = -5.0f, upperY = 25.0f;

gluOrtho2D(lowerX, upperX, lowerY, upperY);

If your game must work in pixel units then you should convert your length units from pixels to meters when passing values from Box2D. Likewise you should convert the values received from Box2D from meters to pixels. This will improve the stability of the physics simulation.

You have to come up with a reasonable conversion factor. I suggest making this choice based on the size of your characters. Suppose you have determined to use 50 pixels per meter (because your character is 75 pixels tall). Then you can convert from pixels to meters using these formulas:

xMeters = 0.02f \* xPixels;

yMeters = 0.02f \* yPixels;

In reverse:

xPixels = 50.0f \* xMeters;

yPixels = 50.0f \* yMeters;

You should consider using MKS units in your game code and just convert to pixels when you render. This will simplify your game logic and reduce the chance for errors since the rendering conversion can be isolated to a small amount of code.

If you use a conversion factor, you should try tweaking it globally to make sure nothing breaks. You can also try adjusting it to improve stability.

# Debug Drawing

You can implement the b2DebugDraw class to get detailed drawing of the physics world. Here are the available entities:

* shape outlines
* joint connectivity
* broad-phase axis-aligned bounding boxes (AABBs)
* center of mass



This is the preferred method of drawing these physics entities, rather than accessing the data directly. The reason is that much of the necessary data is internal and subject to change.

The testbed draws physics entities using the debug draw facility and the contact listener, so it serves as the primary example of how to implement debug drawing as well as how to draw contact points.

# Limitations

Box2D uses several approximations to simulate rigid body physics efficiently. This brings some limitations.

Here are the current limitations:

1. Stacking heavy bodies on top of much lighter bodies is not stable. Stability degrades as the mass ratio passes 10:1.
2. Chains of bodies connected by joints may stretch if a lighter body is supporting a heavier body. For example, a wrecking ball connect to a chain of light weight bodies may not be stable. Stability degrades as the mass ratio passes 10:1.
3. There is typically around 0.5cm of slop in shape versus shape collision.
4. Continuous collision does not handle joints. So you may see joint stretching on fast moving objects.
5. Box2D uses the symplectic Euler integration scheme. It does not reproduce parabolic motion of projectiles and has only first-order accuracy. However it is fast and has good stability.
6. Box2D uses an iterative solver to provide real-time performance. You will not get precisely rigid collisions or pixel perfect accuracy. Increasing the iterations will improve accuracy.

# References

Erin Catto’s GDC Tutorials: <http://code.google.com/p/box2d/downloads/list>

Collision Detection in Interactive 3D Environments, Gino van den Bergen, 2004

Real-Time Collision Detection, Christer Ericson, 2005