Box2D v2.2.1 User Manual

Copyright © 2007-2011 Erin Catto

[Chapter 1 Introduction 2](#_Toc258082967)

[Chapter 2 Hello Box2D 7](#_Toc258082968)

[Chapter 3 Common 13](#_Toc258082969)

[Chapter 4 Collision Module 15](#_Toc258082970)

[Chapter 5 Dynamics Module 22](#_Toc258082971)

[Chapter 6 Fixtures 23](#_Toc258082972)

[Chapter 7 Bodies 26](#_Toc258082973)

[Chapter 8 Joints 33](#_Toc258082974)

[Chapter 9 Contacts 42](#_Toc258082975)

[Chapter 10 World Class 49](#_Toc258082976)

[Chapter 11 Loose Ends 55](#_Toc258082977)

[Chapter 12 Debug Drawing 57](#_Toc258082978)

[Chapter 13 Limitations 59](#_Toc258082979)

[Chapter 14 References 60](#_Toc258082980)

# Introduction

## About关于

Box2D is a 2D rigid body simulation library for games. Programmers can use it in their games to make objects move in believable ways and make the game world more interactive. From the game engine's point of view a physics engine is just a system for procedural animation.

Box2D是一个针对游戏的2D钢体模拟程序库.程序员能够在可信的方式下在他们的游戏中使用Box2D, 并且使游戏有更多的交互. 从游戏引擎的观点来看, 物理引擎就是一个程序式动画.

Box2D is written in portable C++. Most of the types defined in the engine begin with the b2 prefix. Hopefully this is sufficient to avoid name clashing with your game engine.

Box2D使用轻便的C++语言编写. 绝大部分在引擎中的类型都以”b2”为前缀. 希望这足够去避免和你的游戏引擎明明冲突.

## Prerequisites预备知识

In this manual I'll assume you are familiar with basic physics concepts, such as mass, force, torque, and impulses. If not, please first consult Google search and Wikipedia.

在这个手册中, 我将将定你熟悉了基本的物理概念, 例如质量, 力, 转矩和冲量.如果不是, 请首先咨询Google搜索或者wiki.

转矩(转动力矩): <http://zh.wikipedia.org/wiki/%E5%8A%9B%E7%9F%A9>

冲量: <http://zh.wikipedia.org/wiki/%E5%86%B2%E9%87%8F>

Box2D was created as part of a physics tutorial at the Game Developer Conference. You can get these tutorials from the download section of box2d.org.

Box2D在游戏开发者大会上被作为物理辅导的一部分. 你能够从[www.box2d.org](http://www.box2d.org)上获得这些教程.

Since Box2D is written in C++, you are expected to be experienced in C++ programming. Box2D should not be your first C++ programming project. You should be comfortable with compiling, linking, and debugging.

因为Box2D是用C++所写的, 所以期望你对C++编程有一定的经验. Box2D不是你第一个C++编程项目. 你应该先熟练编译, 链接和测试.

**Caution**

Box2D should not be your first C++ project. Please learn C++ programming, compiling, linking, and debugging before working with Box2D. There are many resources for this on the net.

## About this Manual关于这个手册

This manual covers the majority of the Box2D API. However, not every aspect is covered. You are encouraged to look at the testbed included with Box2D to learn more. Also, the Box2D code base has comments formatted for Doxygen, so it is easy to create a hyper-linked API document.

这个手册覆盖了大多数的Box2D API. 然而, 并不是每一个点都覆盖到了. 鼓起勇气去查看Box2D文件夹下的testbed文件内容, 去学到更多的知识点. 并且 Box2D代码基于Doxygen的注解格式, 所以他很容易生成超文本链接API文档.

This manual is only updated with new releases. The version in source control is likely to be out of date.

文档手册只适用于新的release版本. 这个版本可能不再处于源码管理的数据中.

## Feedback and Reporting Bugs反馈和Bugs汇报

If you have a question or feedback about Box2D, please leave a comment in the forum. This is also a great place for community discussion.

如果你有关于Box2D的问题需要反馈, 那么请在论坛中留下注释.对于社区讨论这也是一个伟大的地方.

Box2D issues are tracked using a Google code project. This is a great way to track issues and ensures that your issue will not be lost in the depths of the forums.

Box2D的问题用Google code 项目跟踪. 这是一个非常好的问题跟踪方式, 同时你的问题将不会迷失在论坛的深处.

Please file bugs and feature requests here: <http://code.google.com/p/box2d/>

You can help to ensure your issue gets fixed if you provide sufficient detail. A testbed example that reproduces the problem is ideal. You can read about the testbed later in this document.

如果你提供了足够的细节, 你能够帮助修复你所确定的问题. 一个测试示例就能完美的呈现问题.稍后你将在这篇文档中阅读关于testbed的内容.

## Core Concepts核心概念

Box2D works with several fundamental objects. We briefly define these objects here and more details are given later in this document.

Box2D的工作使用了若干个对象. 我们在这里简单的定义了这些对象, 更多的细节会在后面说到.

#### shape

A 2D geometrical object, such as a circle or polygon.

一个2D多边形对象, 就像一个圆或者多边形.

#### rigid body

A chunk of matter that is so strong that the distance between any two bits of matter on the chunk is constant. They are hard like a diamond. In the following discussion we use body interchangeably with rigid body.

一个大块的物质, 非常坚硬, 物质间任意两点的距离在块体上都是常量. 他们坚硬的就像钻石. 在后面的讨论中我们使用body表示rigid body.

#### fixture

A fixture binds a shape to a body and adds material properties such as density, friction, and restitution.

一个fixture绑定一个shape到一个body上, 同时添加重要的属性, 例如密度, 摩擦力和回弹回复系数.

#### constraint约束

A constraint is a physical connection that removes degrees of freedom from bodies. In 2D a body has 3 degrees of freedom (two translation coordinates and one rotation coordinate). If we take a body and pin it to the wall (like a pendulum) we have constrained the body to the wall. At this point the body can only rotate about the pin, so the constraint has removed 2 degrees of freedom.

一个constraint(约束)是一个物理连接, 能够把自由度从body中去掉. 在2D中, 一个body有3个自由度(两个坐标轴的移动和旋转). 如果我们把一个body钉在墙上(就像钟摆运动), 我们就有了body到墙的约束. 此时此刻body只能够相对于pin旋转, 所以constaint移除了两个自由度.

#### contact constraint接触约束

A special constraint designed to prevent penetration of rigid bodies and to simulate friction and restitution. You do not create contact constraints;they are created automatically by Box2D.

一个特定的constraint有计划的防止钢体穿透, 同时模拟摩擦力和回弹系数. 你不必建立contact constraints; 他们被Box2D自动的创建.

#### joint

This is a constraint used to hold two or more bodies together. Box2D supports several joint types: revolute, prismatic, distance, and more. Some joints may have limits and motors.

这是一个约束, 通常用于保持两个或者更多的body连接. Box2D支持若干个关节类型: 转动(revolute), 移动(prismatic), 距离(distance), 等等. 一些关节可能有limit和motor.

#### joint limit

A joint limit restricts the range of motion of a joint. For example, the human elbow only allows a certain range of angles.

一个joint limit约束一个joint的动作范围. 例如, 一个人类的手肘只允许某一个角度范围.

#### joint motor

A joint motor drives the motion of the connected bodies according to the joint's degrees of freedom. For example, you can use a motor to drive the rotation of an elbow.

一个joint motor驱动连接body的运动, 根据joint的自由度.例如: 你能够使用一个motor驱动手肘的旋转.

#### world

A physics world is a collection of bodies, fixtures, and constraints that interact together. Box2D supports the creation of multiple worlds, but this is usually not necessary or desirable.

物理世界, 是一个bodies, fixtures和constraints一起互动的集合.Box2D支持创造多个世界, 但是这通常是不必要的或者不令人满意的.

#### solver

The physics world has a solver that is used to advance time and to resolve contact and joint constraints. The Box2D solver is a high performance iterative solver that operates in order N time, where N is the number of constraints.

物理世界有一个solver, 通常用于推进时间, 解决碰撞和连接约束. Box2D solver是一个高性能的迭代器, 在顺序的N个时间内操作, N是一个限制数字.

#### continuous collision

The solver advances bodies in time using discrete time steps. Without intervention this can lead to tunneling.

Solver适时的使用离散的时间步推进body前进.在没有干预的情况下, 这会导致贯穿效果.



Box2D contains specialized algorithms to deal with tunneling. First, the collision algorithms can interpolate the motion of two bodies to find the first time of impact (TOI). Second, there is a sub-stepping solver that moves bodies to their first time of impact and then resolves the collision.

Box2D包含专门的算法去解决穿越问题. 首先, 碰撞算法能够在两个运动物体之间差值, 找到第一次发生碰撞的时间(TOI).然后, 有一个子步骤的solver移动物体到他们的第一次碰撞时间, 然后解决碰撞.

## Modules模块

Box2D is composed of three modules: Common, Collision, and Dynamics. The Common module has code for allocation, math, and settings. The Collision module defines shapes, a broad-phase, and collision functions/queries. Finally the Dynamics module provides the simulation world, bodies, fixtures, and joints.

Box2D由三个模块组成: Common, Collision和Dynamics. Common模块适用于内存分配(allocation), 数学处理(math)和设置(settings).Collision模块定义形状(shapes), broad-phase, 和碰撞的函数/问题.最后Dynamics模块提供模拟world, bodies, fixture, 和joint.



## Units单位

Box2D works with floating point numbersand tolerances have to be used to make Box2D perform well. These tolerances have been tuned to work well with meters-kilogram-second (MKS) units. In particular, Box2D has been tuned to work well with moving objects between 0.1 and 10 meters. So this means objects between soup cans and buses in size should work well. Static objects may be up to 50 meters big without too much trouble.

Box2D使用单精度浮点数, 同时公差必须使Box2D表现出色.这些公差已经被调整以米-千克-秒(MKS)为单位来良好的工作. 尤其, Box2D的运动对象已经被调整为**大小**在0.1到10米之间能良好的工作.所以这意味着对象在soup cans和buses尺寸之间应该工作良好. 静态对象能够上升到50米的大小而不会有太多的麻烦.

Being a 2D physics engine, it is tempting to use pixels as your units. Unfortunately this will lead to a poor simulation and possibly weird behavior. An object of length 200 pixels would be seen by Box2D as the size of a 45 story building.

现有的2D物理引擎, 诱导你使用像素作为单位. 不幸的是这将导致一个差劲的模拟和可能怪异的行为. 一个对象200像素长, 在Box2D里将看起来如同45个高楼大小.

**Caution**

Box2D is tuned for MKS units. Keep the size of moving objects roughly between 0.1 and 10 meters. You'll need to use some scaling system when you render your environment and actors. The Box2D testbed does this by using an OpenGL viewport transform. DO NOT USE PIXELS.

It is best to think of Box2D bodies as moving billboards upon which you attach your artwork. The billboard may move in a unit system of meters, but you can convert that to pixel coordinates with a simple scaling factor. You can then use those pixel coordinates to place your sprites, etc.

最好把Box2D bodies看作是移动的布告板, 同时在上面贴上你的原图. 布告板可能在以米为单位的系统中移动, 但是你能够使用简单的缩放因子转换到像素坐标系. 然后你能够使用那些像素坐标去放置你的sprite.

Box2D uses radians for angles. The body rotation is stored in radians and may grow unbounded. Consider normalizing the angle of your bodies if the magnitude of the angle becomes too large (use b2Body::SetAngle).

Box2D使用弧度为角度. body旋转以弧度为单位存储, 同时能够无限的增长.考虑bodies角度的标准化, 如果角度的量级变得太大的话(使用b2Body::SetAngle).

## Factories and Definitions

Memory management plays a central role in the design of the Box2D API. So when you create a b2Bodyor a b2Joint, you need to call the factory functions on b2World. You should never try to allocate these types in another manner.

在Box2D API 设计中, 内存管理扮演一个重要的角色.所以当你建立一个b2Body或者一个b2Joint的时候, 你需要基于b2World调用工厂方法. 你不应该试图分配这些类型在另外一个管理器中.

There are creation functions:

创造函数:

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

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

And there are corresponding destruction functions:

相应的销毁函数:

void b2World::DestroyBody(b2Body\* body)

void b2World::DestroyJoint(b2Joint\* joint)

When you create a body or joint, you need to provide a definition. These definitions contain all the information needed to build the body or joint. By using this approach we can prevent construction errors, keep the number of function parameters small, provide sensible defaults, and reduce the number of accessors.

当你创建一个body或者joint, 你需要提供一个定义.这些定义包括需要被创建的body或者joint的全部的信息. 通过使用这些方法我们能够防止构造错误, 保持函数的参数数量较少, 提供明显的默认值,并且降低访问其数量.

Since fixtures must be parented to a body, they are created and destroyed using a factory method on b2Body:

因为fixture必须以一个body为根, 他们的建立和销毁使用一个b2Body的工厂方法.

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

void b2Body::DestroyFixture(b2Fixture\* fixture)

There is also shortcut to create a fixture directly from the shape and density.

也有快捷方式直接建立一个fixture, 通过shape和density.

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

Factories do not retain references to the definitions. So you can create definitions on the stack and keep them in temporary resources.

工厂方法不为定义保持应用. 所以你能够在堆上创建定义, 同时以临时变量保持他们.

## User Data

The b2Fixture, b2Body, and b2Joint classes allow you to attach user data as a void pointer. This is handy when you are examining Box2D data structures and you want to determine how they relate to the entities in your game engine.

b2Fixture, b2Body, 和b2Joint类允许你以一个空指针依附userData. 当你正在检测Box2D数据结构, 同时你想确定他们如何关联你游戏中实体的时, 这是一个便利的方法.

For example, it is typical to attach an actor pointer to the rigid body on that actor. This sets up a circular reference. If you have the actor, you can get the body. If you have the body, you can get the actor.

例如, 典型的是使一个角色指针关联到钢体, 对那角色也关联钢体. 这是装配一个循环引用. 如果你拥有角色, 你能够获得body. 如果你拥有body, 你能够获得角色.

GameActor\* actor = GameCreateActor();

b2BodyDef bodyDef;

bodyDef.userData = actor;

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

Here are some examples of cases where you would need the user data:

这里列举的一些情况将用使用到userData.

* Applying damage to an actor using a collision result.
* Playing a scripted event if the player is inside an axis-aligned box.
* Accessing a game structure when Box2D notifies you that a joint is going to be destroyed.
* 使用碰撞的结果应用伤害于一个角色上;
* 如果玩家在轴对其的盒子里面, 播放脚本事件.
* 当Box2D通知你一个joint将要被销毁的时候, 将要访问一个游戏结构.

Keep in mind that user data is optional and you can put anything in it. However, you should be consistent. For example, if you want to store an actor pointer on one body, you should keep an actor pointer on all bodies. Don't store an actor pointer on one body, and a foo pointer on another body. Casting an actor pointer to a foo pointer may lead to a crash.

记住userData是一个可选择的并且你能够放置任何东西在里面. 然而, 你需要保持一致性. 例如, 如果你想要存储一个actor指针在一个body上, 你应该保持actor指针在所有的bodies上面. 不要存储一个actor指针在一个body上, 一个foo指针在另外的body上.转换一个actor指针到foo指针可能导致程序崩溃.

User data pointers are NULL by default.

userData指针默认值是NULL.

# Hello Box2D

In the distribution of Box2D is a Hello World project. The program creates a large ground box and a small dynamic box. This code does not contain any graphics. All you will see is text output in the console of the box's position over time.

在Box2D中有一个Hello World的项目. 这个程序创建一个巨大的地板盒子和一个小的动态盒子. 这段代码不包含任何图形. 你将整个时间在控制台看到盒子位置的输出.

This is a good example of how to get up and running with Box2D.

这是一个很好的例子, 介绍如何使用Box2D筹备和运行的.

## Creating a World

Every Box2D program begins with the creation of a b2World object. b2World is the physics hub that manages memory, objects, and simulation. You can allocate the physics world on the stack, heap, or data section.

每个Box2D程序起始于一个b2World对象的构造. b2World是一个物理中心, 管理着内存, 对象,和模拟. 你能够在stack, heap或者数据区域上分配物理世界.

*注:*

*heap：是由malloc之类函数分配的空间所在地。地址是由低向高增长的。*

*stack：是自动分配变量，以及函数调用的时候所使用的一些空间。地址是由高向低减少的。*

It is easy to create a Box2D world. First, we define the gravity vector.Also we tell the world to allow bodies to sleep when they come to rest. A sleeping body doesn't require any simulation.

创建一个Box2D世界是非常简单的. 首先, 我们定义重力向量. 并且我们告诉世界允许bodies可以sleep, 当bodies开始休息的时候. 一个sleep的body不会需要任何的模拟.

b2Vec2 gravity(0.0f, -10.0f);

bool doSleep = true;

Now we create the world object. Note that we are creating the world on the stack, so the world must remain in scope.

现在我们创建world对象. 注意我们在stack中创建world对象, 所以world对象必须在应用范围内被保留.

b2World world(gravity, doSleep);

So now we have our physics world, let's start adding some stuff to it.

现在我们有了我们的物理世界, 开始为他添加一些东西吧.

## Creating a Ground Box

Bodies are built using the following steps:

bodies的创建根据以下步骤:

1. Define a body with position, damping, etc.
2. Use the world object to create the body.
3. Define fixtures with a shape, friction, density, etc.
4. Create fixtures on the body.
5. 定义一个body, 使用位置(position), 阻尼(damping)等参数.
6. 使用世界对象创建一个body.
7. 定义一个fixture, 使用形状(shape), 摩擦力(friction), 密度(density)等.
8. 在body上创建一个fixture.

For step 1 we create the ground body. For this we need a body definition. With the body definition we specify the initial position of the ground body.

通过步骤1,我们创建ground body对象.为此, 我们需要一个body定义对象. 使用body定义对象, 我们指定ground body的初始化位置.

b2BodyDef groundBodyDef;

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

For step 2 the body definition is passed to the world object to create the ground body. The world object does not keep a reference to the body definition. Bodies are static by default. Static bodies don't collide with other static bodies and are immovable.

通过步骤2, body定义对象被传递到world对象, 用来创建一个ground body对象. world对象不会保留body定义对象的引用.bodies默认是静态的. 静态bodies不会与其他静态bodies相互碰撞, 同时是不动的.

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

For step 3 we create a ground polygon. We use the SetAsBox shortcut to form the ground polygon into a box shape, with the box centered on the origin of the parent body.

通过步骤3我们创建了一个ground polygon对象. 我们使用SetAsBox方法快捷的将ground polygon对象构成一个盒子形状, 盒子的中心在根body对象的原心上.

b2PolygonShape groundBox;

groundBox.SetAsBox(50.0f, 10.0f);

The SetAsBox function takes the half-width and half-height (extents). So in this case the ground box is 100 units wide (x-axis) and 20 units tall (y-axis). Box2D is tuned for meters, kilograms, and seconds. So you can consider the extents to be in meters. Box2D generally works best when objects are the size of typical real world objects. For example, a barrel is about 1 meter tall. Due to the limitations of floating point arithmetic, using Box2D to model the movement of glaciers or dust particles is not a good idea.

SetAsBox函数采取一半的宽度和一半的高度. 所以在这种情况下ground box对象是100单位宽和20单位高. Box2D调整为米, 千克, 秒. 所以你能够认为以米为单位的长度. Box2D通常工作良好,在对象的大小是典型的真实世界对象时. 例如, 一个枪管大约1米高. 由于浮点算法的局限性, 使用Box2D创造移动的冰川或者灰尘粒子不是很好的主意.

We finish the ground body in step 4 by creating the shape fixture. For this step we have a shortcut. We do not have a need to alter the default fixture material properties, so we can pass the shape directly to the body without creating a fixture definition. Later we will see how to use a fixture definition for customized material properties. The second parameter is the shape density in kilograms per meter squared. A static body has zero mass by definition, so the density is not used in this case.

在步骤4中通过创建的模型fixture对象,我们完成ground body对象的创建. 对于此次步骤我们有一个捷径. 因为我们不需要修改默认fixture的重要属性, 所以我们能够直接传递shape对象到body, 而没有创建一个fixture定义.一会我们将看到如何使用一个fixture定义对象自定义重要属性.第二个参数是Shape的密度, 以千克每平方米为单位. 一个静态body对象定义重量是0, 所以在这种情况下密度没有被使用.

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

Box2D does not keep a reference to the shape. It clones the data into a new b2Shape object.

Box2D不会保持shape对象的引用. 他只是克隆数据到一个新的b2Shape对象中.

Note that every fixture must have a parent body, even fixtures that are static. However, you can attach all static fixtures to a single static body.

注意, 每一个fixture对象必须有一个根body对象, 即使fixture是静态的. 然而, 你能够依附全部静态fixture到一个单独的静态body对象上.

## Creating a Dynamic Body

So now we have a ground body. We can use the same technique to create a dynamic body. The main difference, besides dimensions, is that we must establish the dynamic body's mass properties.

现在我们有一个ground body对象了. 我们能够使用相同的技术创建一个动态body对象.主要的不同之处, 除了规模, 那就是我们必须设置动态body对象的质量属性.

First we create the body using CreateBody. By default bodies are static, so we should set the b2BodyType at construction time to make the body dynamic.

首先我们建立body对象使用CreateBody函数. 默认的bodies对象是静态的, 所以我们应该在构造时设置b2BodyType为动态的.

b2BodyDef bodyDef;

bodyDef.type = b2\_dynamicBody;

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

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

Caution

You must set the body type to b2\_dynamicBody if you want the body to move in response to forces.

Next we create and attach a polygon shape using a fixture definition.First we create a box shape:

接下来我们创建同时依附一个shape对象到fixture定义对象上.首先我们创建一个盒子形状:

b2PolygonShape dynamicBox;

dynamicBox.SetAsBox(1.0f, 1.0f);

Next we create a fixture definition using the box. Notice that we set density to 1. The default density is zero. Also, the friction on the shape is set to 0.3.

接下来我们使用盒子对象创建一个fixture定义对象. 注意我们设置密度为1.默认密度是0.并且,摩擦力在shape对象上被设置为0.3.

b2FixtureDef fixtureDef;

fixtureDef.shape = &dynamicBox;

fixtureDef.density = 1.0f;

fixtureDef.friction = 0.3f;

Using the fixture definition we can now create the fixture. This automatically updates the mass of the body. You can add as many fixtures as you like to a body. Each one contributes to the total mass.

使用fixture定义对象我们能够创建一个fixture对象. 这会自动化更新body对象质量. 你能添加很多fixture对象到body上. 每一个fixture捐献总质量.

body->CreateFixture(&fixtureDef);

That's it for initialization. We are now ready to begin simulating.

这是他的初始化. 我们现在开始准备模拟了.

## Simulating the World (of Box2D)

So we have initialized the ground box and a dynamic box. Now we are ready to set Newton loose to do his thing. We just have a couple more issues to consider.

我们初始化了ground box对象和一个动态box对象. 现在我们准备让牛顿不精确的做他的事情.我们只有两个问题需要考虑了.

Box2D uses a computational algorithm called an integrator. Integrators simulate the physics equations at discrete points of time. This goes along with the traditional game loop where we essentially have a flip book of movement on the screen. So we need to pick a time step for Box2D. Generally physics engines for games like a time step at least as fast as 60Hz or 1/60 seconds. You can get away with larger time steps, but you will have to be more careful about setting up the definitions for your world. We also don't like the time step to change much. A variable time step produces variable results, which makes it difficult to debug. So don't tie the time step to your frame rate (unless you really, really have to). Without further ado, here is the time step.

Box2D使用一个计算算法调用一个积分器. 积分器在离散的时间点模拟物理方程式. 积分器会附和在传统的游戏循环中, 我们本来在屏幕上就有一个翻书运动. 所以我们需要拾取一个时间步给Box2D. 通常游戏物理引擎至少像60Hz或者1/60秒那样快. 虽然你能够的使用更大的时间步,但是你将必须小心的设置定义你的世界.我们同样不喜欢时间步改变太多. 一个变化的时间步产生变化的结果, 这将导致调试困难. 所以不要把时间步与你的帧率约束到一起(除非你真的真的必须这么做).事不宜迟, 下面是时间步.

float32 timeStep = 1.0f / 60.0f;

In addition to the integrator, Box2D also uses a larger bit of code called a constraint solver. The constraint solver solves all the constraints in the simulation, one at a time. A single constraint can be solved perfectly. However, when we solve one constraint, we slightly disrupt other constraints. To get a good solution, we need to iterate over all constraints a number of times.

除了积分器外, Box2D同样使用一个更大一点的代码调用一个约束求解程序(constraint solver). 约束求解程序解决所有的在模拟中的约束, 每次迭代会迭代每一个约束. 单个约束能够被完美的解决. 然而, 当我们解决一个约束时, 我们轻微的破坏了其他的约束.为了得到一个很好的解决方案, 我们需要多次迭代全部的约束.

There are two phases in the constraint solver: a velocity phase and a position phase. In the velocity phase the solver computes the impulses necessary for the bodies to move correctly. In the position phase the solver adjusts the positions of the bodies to reduce overlap and joint detachment. Each phase has its own iteration count. In addition, the position phase may exit iterations early if the errors are small.

约束求解程序有两个阶段: 速度阶段和位置阶段. 在速度阶段解决计算冲量, 是对bodies对象的正确移动是有必要的. 在位置阶段解决调整bodies的位置, 减少重叠和joint分离. 每个阶段都有自己的迭代次数.另外, 位置阶段可能会较早的退出迭代, 如果误差是比较小的话.

The suggested iteration count for Box2D is 8 for velocity and 3 for position. You can tune this number to your liking, just keep in mind that this has a trade-off between speed and accuracy. Using fewer iterations increases performance but accuracy suffers. Likewise, using more iterations decreases performance but improves the quality of your simulation. For this simple example, we don't need much iteration. Here are our chosen iteration counts.

对于Box2D建议的迭代次数, 速度迭代是8, 位置迭代是3. 你能够调整为你喜欢的数字, 切记, 这是在速度和精确度之间的权衡. 使用更少的迭代增加执行效率, 但是精确度降低. 同样的, 使用更高的迭代降低运行效率, 但是改善模拟的质量. 举一个简单的例子, 我们不需要太多的迭代. 下面是我们选择的迭代次数.

int32 velocityIterations = 6;

int32 positionIterations = 2;

Note that the time step and the iteration count are completely unrelated. An iteration is not a sub-step. One solver iteration is a single pass over all the constraints within a time step. You can have multiple passes over the constraints within a single time step.

注意: 时间步和迭代次数是完全无关的. 一个迭代并不是一个时间步的子步.一次迭代就是在时间步之中遍历所有约束. 你可以在单个时间步内遍历多次.

We are now ready to begin the simulation loop. In your game the simulation loop can be merged with your game loop. In each pass through your game loop you call b2World::Step. Just one call is usually enough, depending on your frame rate and your physics time step.

我们现在准备开始循环模拟. 在你的游戏中, 模拟循环能够与你的游戏循环相结合. 在每次经过你游戏循环的时候调用b2World::Step方法. 通常调用一次就足够了, 不过这依赖于你的帧率和物理时间步.

The Hello World program was designed to be simple, so it has no graphical output. The code prints out the position and rotation of the dynamic body. Here is the simulation loop that simulates 60 time steps for a total of 1 second of simulated time.

Hello World程序设计的非常简单, 所以他没有图形输出. 代码打印出动态body的位置和旋转角度.下面是模拟循环60个时间步, 1秒钟的时间模拟.

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);

}

The output shows the box falling and landing on the ground box. Your output should look like this:

输出显示盒子坠落并且着陆在地面上. 你的输出应该看起来类似这样:

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

## Cleanup

When a world leaves scope or is deleted by calling delete on a pointer, all the memory reserved for bodies, fixtures, and joints is freed. This is done to improve performance and make your life easier. However, you will need to nullify any body, fixture, or joint pointers you have because they will become invalid.

当一个world对象离开作用域或者在一个指针上被删除, 所有的保留的bodies, fixtures, joints内存被释放. 这样做是改善执行效率同时使你的生活更加简单.然而, 你将需要作废每一个body, fixture, joint对象, 因为他们已经变为无效的了.

## The Testbed

Once you have conquered the HelloWorld example, you should start looking at Box2D's testbed. The testbed is a unit-testing framework and demo environment. Here are some of the features:

当你在Hello World程序中取得胜利之后, 你应该开始查看Box2D的testbed文件夹里的内容.testbed是一个单元测试框架, 同时也是一个演示环境. 下面介绍了一些特征:

* Camera with pan and zoom.
* Mouse picking of shapes attached to dynamic bodies.
* Extensible set of tests.
* GUI for selecting tests, parameter tuning, and debug drawing options.
* Pause and single step simulation.
* Text rendering.
* 镜头摇动和缩放.
* 鼠标拾取Shape依附于动态body.
* 可扩展的测试集合.
* 用户测试选择界面, 参数调整, 调试绘制选项.
* 暂停并且单步模拟.
* 文本渲染.



The testbed has many examples of Box2D usage in the test cases and the framework itself. I encourage you to explore and tinker with the testbed as you learn Box2D.

在testbed文件夹中有很多在测试中Box2D惯用的例子, 和框架本身的例子.我鼓励你去探索并且修改testbed内容作为你学习Box2D的起点.

Note: the testbed is written using freeglut and GLUI. The testbed is not part of the Box2D library. The Box2D library is agnostic about rendering. As shown by the HelloWorld example, you don't need a renderer to use Box2D.

注意: testbed使用freeglut和GLUI编写.testbed不是Box2D库函数的一部分. Box2D库对于渲染来说是不可知的. 如同HelloWorld例子所示, 使用Box2D你可以不使用渲染.

# Common

## About

The Common module contains settings, memory management, and vector math.

Common模块包含设置, 内存管理和向量数学.

## Settings

The header b2Settings.h contains:

b2Setting.h头文件包含:

* Types such as int32 and float32
* Constants
* Allocation wrappers
* The version number
* 类型, 比如: int32和float32.
* 常量.
* 内存分配包装器.
* 版本号.

### Types

Box2D defines various types such as float32, int8, etc. to make it easy to determine the size of structures.

Box2D定义了变量类型例如float32, int8, 等. 为了更简单的决定数据结构的大小.

### Constants

Box2D defines several constants. These are all documented in b2Settings.h. Normally you do not need to adjust these constants.

Box2D定义了若干个常量. 所有的文档在b2Settings.h文件里面. 通常你不需要去调整这些常量.

Box2D uses floating point math for collision and simulation. Due to round-off error some numerical tolerances are defined. Some tolerances are absolute and some are relative. Absolute tolerances use MKS units.

Box2D为碰撞和模拟使用浮点算数符. 由于四舍五入错误, 定义了一些数值公差. 一些公差是绝对的, 一些公差是相对的.绝对公差使用MKS(米/千克/秒)为单位.

### Allocation wrappers

The settings file defines b2Alloc and b2Free for large allocations. You may forward these calls to your own memory management system.

Settings文件定义了b2Alloc和b2Free函数, 他们适用于大多数内存分配. 你可以使用这些函数, 在你自己的内存管理系统中.

### Version

The b2Version structure holds the current version so you can query this at run-time.

b2Version结构持有当前版本, 所以你能够在运行时查询他.

## Memory Management

A large number of the decisions about the design of Box2D were based on the need for quick and efficient use of memory. In this section I will discuss how and why Box2D allocates memory.

对于Box2D的设计有着大量的讨论, 他们都是基于需要快速和有效率的内存使用. 在这一章, 我将讨论Box2D如何和为什么分配内存.

Box2D tends to allocate a large number of small objects (around 50-300 bytes). Using the system heap through malloc or new for small objects is inefficient and can cause fragmentation. Many of these small objects may have a short life span, such as contacts, but can persist for several time steps. So we need an allocator that can efficiently provide heap memory for these objects.

Box2D有益于分配大多数小对象(大概50 – 300字节). 通过使用系统堆(heap)malloc或者new生成一个小对象是很没有效率的, 同时会引起存储残片. 许多这些小对象可能只有一个短生命周期, 例如:联系(contacts), 但是能够持续若干个时间周期. 所以我们需要一个分配器, 他能够为这些对象有效率的提供堆(heap)内存.

Box2D's solution is to use a small object allocator (SOA) called b2BlockAllocator. The SOA keeps a number of growable pools of varying sizes. When a request is made for memory, the SOA returns a block of memory that best fits the requested size. When a block is freed, it is returned to the pool. Both of these operations are fast and cause little heap traffic.

Box2D的解决方法是使用一个小对象分配器(SOA), 通过调用b2BlockAllocator.SOA保留一个许多大小不同可增长的池. 在请求一个内存的时候, SOA返回一个最适合请求大小的内存块.当一个块被释放的时候, 块被返回到池中. 这两个操作是非常迅速的, 并且只会引起一点点的堆(heap)流量.

Since Box2D uses a SOA, you should never new or malloc a body, fixture, or joint. However, you do have to allocate a b2World on your own. The b2World class provides factories for you to create bodies, fixtures, and joints. This allows Box2D to use the SOA and hide the gory details from you. Never, call delete or free on a body, fixture, or joint.

因为Box2D使用一个SOA, 所以你不应该使用new或者malloc生成一个body, fixture, joint对象.然而, 必须由你自己负责分配一个b2World对象. b2World类提供工厂方法创建bodies, fixtures, joints.这允许Box2D使用SOA和对你隐藏血淋淋的细节.你永远不要delete 或者freebody, fixture, joint对象.

While executing a time step, Box2D needs some temporary workspace memory. For this, it uses a stack allocator called b2StackAllocator to avoid per-step heap allocations. You don't need to interact with the stack allocator, but it's good to know it's there.

当前在执行一个时间步的时候, Box2D需要一些临时工作内存. 因此, 使用一个stack分配器调用b2StackAllocator避免每一步的heap分配. 虽然你不要与stack分配器相互作用, 但是你对他有所了解还是很好的.

## Math

Box2D includes a simple small vector and matrix module. This has been designed to suit the internal needs of Box2D and the API. All the members are exposed, so you may use them freely in your application.

Box2D包含一个简单的小向量和矩阵模块. 这些都被设计为适合Box2D的内部需要, 和API的使用. 因为所有成员都被暴露, 所以你可以在你的应用程序中更自由的使用他们.

The math library is kept simple to make Box2D easy to port and maintain.

数学库保持简单, 易于移植和维护.

# Collision Module

## About

The Collision module contains shapes and functions that operate on them. The module also contains a dynamic tree and broad-phase to acceleration collision processing of large systems.

碰撞模块包含Shapes和在他们上面操作的功能. 模型同时包含一个动态树和一个broad-phase, 加速在大系统中的碰撞处理.

The collision module is designed to be usable outside of the dynamic system. For example, you can use the dynamic tree for other aspects of your game besides physics.

碰撞模块被设计于在外部系统上可用. 例如, 你能够在你游戏除了物理的其他方面使用动态树.

## Shapes

Shapes describe collision geometry and may be used independently of physics simulation. You may perform several operations with shapes.

Shapes描述碰撞形状, 也可以单独使用物理模拟. 你可以执行对模型执行若干操作.

Box2D shapes implement the b2Shape base class. The base class defines functions to:

Box2D 的Shapes以b2Shape为基类实现.基类定义的功能:

* Test a point for overlap with the shape.
* Perform a ray cast against the shape.
* Compute the shape's AABB.
* Compute the mass properties of the shape.
* 测试一个点与shape是否重叠.
* 执行射线投影在shape.
* 计算shape的AABB.
* 计算shape的质量属性.

In addition, each shape has a type member and a radius. The radius even applies to polygons, as discussed below.

另外, 每一个shape有一个类型(type)成员和半径(radius). 半径甚至作用于多边形, 讨论如下.

## Circle Shapes

Circle shapes have a position and radius.

圆有位置和半径.

Circles are solid. You cannot make a hollow circle. However, you can create chains of line segments using polygon shapes.

圆是实心的.你不能制造一个空心的圆. 然而, 你能使用多边形创建线段链.

b2CircleShape circle;

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

circle.m\_radius = 0.5f;

## Polygon Shapes

Polygon shapes are solid convex polygons. A polygon is convex when all line segments connecting two points in the interior do not cross any edge of the polygon. Polygons are solid and never hollow. A polygon must have 3 or more vertices.

多边形是实心凸多边形. 一个多边形当在内部两点连接的所有线段不与多边形的任何边交叉的时候, 那么这个多边形就是个凸多边形.多边形是实心且永远不会是空心的. 一个多边形必须有3个以上的顶点.



You must create polygons with a counter clockwise winding (CCW). We must be careful because the notion of CCW is with respect to a right-handed coordinate system with the z-axis pointing out of the plane.This might turn out to be clockwise on your screen, depending on your coordinate system conventions.

你必须以逆时针方式创建多边形. 我们必须小心, 因为逆时针概念遵守的是右手坐标系统, z轴指向平面外.有可能相对于你的屏幕就编程顺时针的了, 这取决于你坐标系统的协定.



The polygon members are public, but you should use initialization functions to create a polygon. The initialization functions create normal vectors and perform validation.

多边形成员是公共的, 但是你应该使用函数初始化并且建立一个多边形.初始化函数创建普通向量和执行生效.

You can create a polygon shape by passing in a vertex array. The maximal size of the array is controlled by b2\_maxPolygonVerticeswhich has a default value of 8. This is sufficient to describe most convex polygons.

你能够创建一个多边形Shape, 通过传递一个顶点数组. 数组最大的大小为b2\_maxPolygonVerices,默认值为8. 这足够描述大多数凸多边形了.

// 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);

The polygon shape has some custom initialization functions to create boxes.

多边形Shapes有一些自定义初始化函数用来创建盒子.

void SetAsBox(float32 hx, float32 hy);

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

Polygons inherit a radius from b2Shape. The radius creates a skin around the polygon. The skin is used in stacking scenarios to keep polygons slightly separated. This allows continuous collision to work against the core polygon.

多边形从b2Shape继承一个 radius(半径) 属性. radius(半径)建立一个 skin(外皮)围绕着多边形. Skin(外皮)用于在堆积情况下保持多边形少量的分离.这允许连续碰撞对核心多边形的分离.­



The polygon skin helps prevent tunneling by keeping the polygons separated. This results in small gaps between the shapes. Your visual representation can be larger than the polygon to hide any gaps.

多边形皮肤通过保持多边形的分离, 帮助防止隧道效应(tunneling). 这会导致在shape间存在一个很小的间隙. 你的可视化表示能够大于多边形, 以隐藏任何一个间隙.



## Edge Shapes(边缘Shape)

Edge shapes are line segments. These are provided to assist in making a free-form static environment for your game. A major limitation of edge shapes is that they can collide with circles and polygons but not with themselves. The collision algorithms used by Box2D require that at least one of two colliding shapes have volume. Edge shapes have no volume, so edge-edge collision is not possible.

边形状(Edge Shape)是一组线段. 提供帮助你的游戏创造一个自由形态的静态环境. 边界Shape的主要限制是他们能够与圆和多边形发生碰撞, 但是他们彼此之间却不行.根据Box2D碰撞算法的要求, 至少有两个碰撞的shapes体积. 边shapes没有体积, 所以边边碰撞检测是没有可能的.

// This an edge shape.

b2Vec2 v1(0.0f, 0.0f);

b2Vec2 v2(1.0f, 0.0f);

b2EdgeShape edge;

edge.Set(v1, v2);

In many cases a game environment is constructed by connect several edge shapes end-to-end. This can give rise to an unexpected artifact when a polygon slides along the chain of edges. In the figure below we see a box colliding with an internal vertex. These *ghost* collisions are caused when the polygon collides with an internal vertex generating an internal collision normal.

在多种情况下, 一个游戏环境通过若干个首尾相连的边shapes创建而成. 当一个多边形沿着边界链(chain of edges)滑动时, 这能够引起意想不到的假象. 如下图所示, 我们看见一个盒子与一个内部顶点碰撞. 幽灵碰撞(ghost collision)的触发是: 当多边形和一个内部顶点碰撞, 产生一个正常的内部碰撞.



If edge1 did not exist this collision would seem fine. With edge1 present, the internal collision seems like a bug. But normally when Box2D collides two shapes, it views them in isolation.

如果edge1不存在碰撞将会看起来很好. 就edge1现状, 内部碰撞看起来像个Bug. 但是通常当Box2D的两个shapes碰撞, Box2D考虑分离他们.

Fortunately, the edge shape provides a mechanism for eliminating ghost collisions by storing the adjacent *ghost* vertices. Box2D uses these ghost vertices to prevent internal collisions.

幸运的, Edge shape提供一个技巧以消除幽灵碰撞(ghost collision), 通过保管临近的幽灵顶点(ghost vertices). Box2D使用这些幽灵顶点(ghost collision)防止内部碰撞.



// 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;

In general stitching edges together this way is a bit wasteful and tedious. This brings us to the next shape type.

一般而言缝合各个边这个方法是有一点浪费和乏味的. 所以这带给我们下面的一种类型.

## Chain Shapes

The chain shape provides an efficient way to connect many edges together to construct your static game worlds. Chain shapes automatically 1 ghost collisions and provide two sided collision.

链式形状(chain shape)提供一个有效率的方式连接许多的边, 用来创建一个游戏世界. 链shape自动消除幽灵碰撞(ghost collision), 并且提供两边界碰撞.



// 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);

You may have a scrolling game world and would like to connect several chains together. You can connect chains together using ghost vertices, like we did with b2EdgeShape.

你可能有一个滚动的游戏世界, 同时相互连接几个链. 你能够使用幽灵顶点(ghost vertices)相互连接链, 就像我们使用b2EdgeShape所做的一样.

// Install ghost vertices

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

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

You may also create loops automatically.

你也能够自动的建立一个循环.

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

b2ChainShape chain;

chain.CreateLoop(vs, 4);

Self-intersection of chain shapes is not supported. It might work, it might not. The code that prevents ghost collisions assumes there are no self-intersections of the chain.

链shapes的自相交是不支持的. 它可能会工作, 也可能不会. 代码假定没有链的自相交, 防止幽灵碰撞(ghost collision).



Each edge in the chain is treated as a child shape and can be accessed by index.

每一条边在链中看作是一个子shape, 并且能够通过索引访问.

// Visit each child edge.

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

{

b2EdgeShape edge;

chain.GetChildEdge(&edge, i);

…

}

## In Shape Point Test(形状的点测试)

You can test a point for overlap with a shape. You provide a transform for the shape and a world point.

你能够测试一个点与shape的重叠. 你提供一个shape的变换(transform)和一个世界坐标.

b2Transfrom transform;

transform.SetIdentity();

b2Vec2 point(5.0f, 2.0f);

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

Edge and chain shapes always return false, even if the chain is a loop.

Edge和chain shapes总是返回false, 即使链shape是循环结构.

## Shape Ray CastShape光线投射

You can cast a ray at a shape to get the point of first intersection and normal vector. No hit will register if the ray starts inside the shape. A child index is included for chain shapes because the ray cast will only check a single edge at a time.

你能够投射一道光线在shape上, 以获得第一个相交的点, 和一个法相向量.如果射线在shape的内部, 将没有碰撞将被记录. 链shape的子索引会被包含, 因为射线投影每次只检测单个边.

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);

…

}

(译注: 这里说的光线指几何中的射线。

看看b2RayCastInput的定义，除指定了两个点p1, p2外，还有个maxFraction。这个maxFraction是什么意思呢? 我们知道，两点决定一个直线，在数学上知道了两点，再定义直线上的其它点，常使用参数方程。也就是定义 P(fraction) = p1 + fraction \* (p2 - p1)。当fraction为0时，就代表p1, 当fraction=1时，就代表p2。这样的定义下，两点之间的线段就是参数从0到1之间变化。参数小于0，表示反向的点，大于1就表示正向超出线段的点。maxFraction就表示要测试的点对应的参数是在[0, maxFraction]内。b2RayCastOutput也有个fraction，意思是一样的。

数学上很喜欢将一些变量归结成0到1之间变化，这叫做规范化。处理问题的常用手段是用某个变换(这里说的变换是广义的)将变量归结成0到1之间，再在规范化之下计算, 之后再用个反变换得到原问题的答案。上面说的直线参数化，可以看成规范化的一种。那为什么要规范化呢？因为这样计算起来方便。那为什么会方便呢？我就答不出来了。Box2D中凡是涉及到向量的，那个单词fraction应该都是上面说的意思。）

## Bilateral Functions双向功能

The Collision module contains bilateral functions that take a pair of shapes and compute some results. These include:

碰撞模块包含一些双向功能, 获得一对shapes并且计算一些结果.

包含这些:

* Overlap
* Contact manifolds
* Distance
* Time of impact
* 重叠
* 接触manifolds
* 距离
* 碰撞时间

## Overlap

You can test two shapes for overlap using this function:

你能够使用下面的函数测试两个shape的重叠.

b2Transform xfA = …, xfB = …;

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

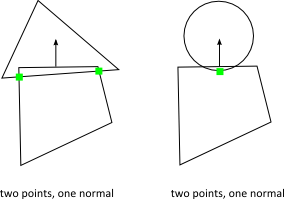
Again you must provide child indices to for the case of chain shapes.

链shape在这种情况下必须提供子索引.

## Contact Manifolds

Box2D has functions to compute contact points for overlapping shapes. If we consider circle-circle or circle-polygon, we can only get one contact point and normal. In the case of polygon-polygon we can get two points. These points share the same normal vector so Box2D groups them into a manifold structure. The contact solver takes advantage of this to improve stacking stability.

Box2D拥有功能计算重叠shape的接触点. 如果我们考虑缘圆和圆, 或者圆和多边形, 我们只能获得一个接触点和法向量.在多边形和多边形的情况下, 我们能够获得两个点. 这些点共享相同的法向量, 所以Box2D把他们分成一个manifold数据结构. 接触解决器(contact solver)采取这种优势改善堆积(stacking)的稳定性.



Normally you don’t need to compute contact manifolds directly, however you will likely use the results produced in the simulation.

通常情况下, 你不需要直接计算接触manifold, 不管怎样, 你将很有可能使用在模拟中处理过的结果.

The b2Manifold structure holds a normal vector and up to two contact points. The normal and points are held in local coordinates. As a convenience for the contact solver, each point stores the normal and tangential (friction) impulses.

b2Manifold数据结构保持一个法向量, 并且取决于两个接触点. 法向量和点被保留在局部坐标系. 为了方便接触解决器(contact solver), 每个点存储法向量和切线冲量(摩擦力).

The data stored in b2Manifold is optimized for internal use. If you need this data, it is usually best to use the b2WorldManifold structure to generate the world coordinates of the contact normal and points. You need to provide a b2Manifold and the shape transforms and radii.

数据存储在b2Manifold中, 被优化于内部使用. 如果你需要这些数据, 通常最好使用b2WorldManifold结构产生一个世界坐标系的法向量和点.你需要提供一个b2Manifold和shapes的变形与半径.

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];

…

}

During simulation shapes may move and the manifolds may change. Points may be added or removed. You can detect this using b2GetPointStates.

在模拟期间, shapes可能移动, 同时manifolds可能改变.点可能会被添加或者移除. 你能够用b2GetPointStates来检测.

b2PointState state1[2], state2[2];

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

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

{

// process event

}

## Distance

The b2Distance function can be used to compute the distance between two shapes. The distance function needs both shapes to be converted into a b2DistanceProxy. There is also some caching used to warm start the distance function for repeated calls. You can see the details in b2Distance.h.

b2Dstance函数能够被用于计算两个shapes之间的距离.距离函数需要两个shapes转变为一个b2DistanceProxy. 也有一些缓存用于热启动重复调用距离函数.你能够在b2Distance.h看到详细信息.



## Time of Impact

If two shapes are moving fast, they may *tunnel* through each other in a single time step.

如果两个shapes移动过快, 他们在一个单独的时间步可以互相穿越.



The b2TimeOfImpact is used to determinethe time when two moving shapes collide. This is called the *time of impact* (TOI). The main purpose of b2TimeOfImpact is for tunnel prevention. In particular, it is designed to prevent moving objects from tunneling outside of static level geometry.

当两个移动的shapes碰撞时, b2TimeOfImpact用于决定碰撞的时间. 叫做碰撞时间*time of impact*(TOI). b2TimeOfImpact主要的用途防止穿越(隧道效应). 特别是, 它设计防止运动物体穿过静态几何平面的外面.

Thisfunction accounts for rotation and translation of both shapes, however if the rotations are large enough, then the function may miss a collision. However the function will still report a non-overlapped time and will capture all translational collisions.

函数对shapes的旋转和变形负有责任, 可是如果旋转足够大, 那么函数可能丢失一个碰撞.不管怎样函数仍将报告一个非重叠时间, 并且将捕获所有平移的碰撞.

The time of impact function identities an initial separating axis and ensures the shapes do not cross on that axis. This will miss collisions that are clear at the final positions. While this approach may miss some collisions, it is very fast and adequate for tunnel prevention.

碰撞时间的函数特性是标识一个初始分离轴同时确定shapes与轴线没有相交.不过这将会在最后的位置上完全错过碰撞. 虽然这个方法可能错过一些碰撞, 但他运行的非常快同时足够用于预防穿越.





It is difficult to put a restriction on the rotation magnitude. There may be cases where collisions are missed for small rotations. Normally, these missed rotational collisions should not harm game play.

在旋转的量级上很难赋予一个限制. 可能在大多情况下, 碰撞会忽略小的碰撞. 通常, 这些忽略的旋转碰撞不会伤害游戏性.

The function requires two shapes (converted to b2DistanceProxy) and two b2Sweep structures. The sweep structure defines the initial and final transforms of the shapes.

初始和最后的shapes变形.函数需要两个shapes(转变为b2DistanceProxy)和两个b2Sweep数据结构. (扫描)sweep数据结构定义了

You can use fixed rotations to perform a *shape cast*. In this case, the time of impact function will not miss any collisions.

你能够使用固定旋转角执行一个形状投影(shape cast). 在这种情况下, 碰撞时间函数将不会错过任何碰撞.

## Dynamic Tree

The b2DynamicTree class is used by Box2D to organize large numbers of shapes efficiently. The class does not know about shapes. Instead it operates on axis-aligned bounding boxes (AABBs) with user data pointers.

Box2D使用b2DynamicTree类高效率的组织shapes.这个类不需要知道shape的任何信息. 反而它使用用户数据指针对轴包围盒做操作(axis-aligned bounding boxes, AABBs).

The dynamic tree is a hierarchical AABB tree. Each internal node in the tree can has two children. A leaf node is a single user AABB. The tree uses rotations to keep the tree balance, even in the case of degenerate input.

动态树是一个分层的AABB树. 树中的每一个内部节点能够有两个子节点. 一个叶子节点是一个单一的用户AABB. 这颗树使用旋转保持树的平衡, 甚至在恶劣输入情况下.

The tree structure allows for efficient ray casts and region queries. For example, you may have hundreds of shapes in your scene. You could perform a ray cast against the scene in a brute force manner by ray casting each shape. This would be inefficient because it does not take advantage of shapes being spread out. Instead, you can maintain a dynamic tree and perform ray casts against the tree. This traverses the ray through the tree skipping large numbers of shapes.

树的数据结构允许高效的射线投影和区域查询. 例如: 在你的场景中可能有成百上千个shapes.你能够以一种暴力的方式执行一个射线投影到场景, 投影到每个shape上. 这将是低效率的, 因为它没有对分摊开的shapes采取优化处理. 反而, 你能够维持一个动态树和执行射线投影到树上. 遍历射线穿过的树, 跳过大量的shapes.

A region query uses the tree to find all leaf AABBs that overlap a query AABB. This is faster than a brute force approach because many shapes can be skipped.

范围查询使用树获得与查询AABB有重叠的所有叶子节点AABBs. 这会比暴力途径来的快, 因为很多shapes能够被跳过.





Normally you will not use the dynamic tree directly. Rather you will go through the b2World class for ray casts and region queries. If you do plan to create your own dynamic tree, you can learn how to use it by looking at how Box2D uses it.

通常你将不会直接的使用动态树. 代替的是你会通过b2World类获得射线投影和区域查询. 如果你计划创建一个你自己的动态, 你可以通过查看Box2D学会如何使用他.

## Broad-phase

Collision processing in a physics step can be divided into narrow-phase and broad-phase. In the narrow-phase we compute contact points between pairs of shapes. Imagine we have N shapes. Using brute force, we would need to perform the narrow-phase for N\*N/2 pairs.

碰撞处理在物理阶段能够被划分为两个阶段, narrow-phase和broad-phase. 在narrow-phase我们计算shapes之间的接触点. 想像我们有N个shapes. 使用强力查询, 我们需要执行narrow-phaseN\*N/2次.

The b2BroadPhase class reduces this load by using a dynamic tree for pair management. This greatly reduces the number of narrow-phase calls.

b2BroadPhase类减少这个负载, 通过使用一个动态树成对管理. 这可以大大的减少narrow-phase的调用次数.

Normally you do not interact with the broad-phase directly. Instead, Box2D creates and manages a broad-phase internally. Also, b2BroadPhase is designed with Box2D’s simulation loop in mind, so it is likely not suited for other use cases.

通常情况下你不用直接的与broad-phase相互作用. 代替的是Box2D创建和管理一个内部broad-phase.并且, b2BroadPhase被设计于在内部(in mind)使用Box2D的模拟循环, 所以他很可能不适合于适应其他情况.

# Dynamics Module

## Overview

The Dynamics module is the most complex part of Box2D and is the part you likely interact with the most. The Dynamics module sits on top of the Common and Collision modules, so you should be familiar with those by now.

Dynamics框架是Box2D中最复杂的部分, 同时很可能是你交互最多的. Dynamics框架位于Common和Collision框架之上, 所以你现在应该熟悉他们了.

The Dynamics module contains:

Dynamics框架包含:

* shape fixture class
* rigid body class
* contact class
* joint classes
* world class
* listener classes

There are many dependencies between these classes so it is difficult to describe one class without referring to another. In the following, you may see some references to classes that have not been described yet. Therefore, you may want to quickly skim this chapter before reading it closely.

这些类之间有很多的依赖, 所以很难在其他情况下单独描述一个类. 在下文中, 你可以看见一些还没有被描述的类被谈及.因此, 你可以快速略读这些章节, 之后再精读.

The dynamics module is covered in the following chapters.

Dynamics模块一个覆盖了下面的章节.

# Fixtures

## About

Recall that shapes don’t know about bodies and may be used independently of the physics simulation.Therefore Box2D provides the b2Fixture class to attach shapes to bodies. Fixtures hold the following:

回忆一下, shape不知道body, 同时也被用于独立的物理模拟.因此Box2D提供b2Fixture类依附shape到body. Fixture功能如下所示:

* a single shape
* broad-phase proxies
* density, friction, and restitution
* collision filtering flags
* back pointer to the parent body
* user data
* sensor flag
* 一个单独的shape
* broad-phase代理
* 密度, 摩擦力, 回弹系数
* 碰撞过滤标识
* 指向根body的指针
* 用户数据
* 传感器标志

These are described in the following sections.

在下面的章节中会描述.

## Fixture Creation

Fixtures are created by initializing a fixture definition and then passing the definition to the parent body.

Fixture的创建通过初始化一个FixtureDef然后传递定义给根body对象.

b2FixtureDef fixtureDef;

fixtureDef.shape = &myShape;

fixtureDef.density = 1.0f;

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

This creates the fixture and attaches it to the body. You do not need to store the fixture pointer since the fixture will automatically be destroyed when the parent body is destroyed. You can create multiple fixtures on a single body.

这建立了fixture和依附它到body上. 你不需要存储fixture指针, 因为当根body 销毁的时候fixture将自动被销毁.你能创建多个fixture在一个单独的body上.

You can destroy a fixture on the parent body. You may do this to model a breakable object. Otherwise you can just leave the fixture alone and let the body destruction take care of destroying the attached fixtures.

你能够在根body上销毁一个fixture. 你可以用这个模仿一个碎裂的对象. 另外你可以不管fixture, 并且让body的析构函数的管理要销毁的已经关联的fixture.

myBody->DestroyFixture(myFixture);

### Density

The fixture density is used to compute the mass properties of the parent body. The density can be zero or positive. You should generally use similar densities for all your fixtures. This will improve stacking stability.

Fixture的密度用于计算根body的质量属性. 密度能够是0或者正数. 通常你应该对所有fixtures使用相同的质量. 这将改善堆栈(stacking)的稳定性.

The mass of a body is not adjusted when you set the density. You must call ResetMassData for this to occur.

在你设置density时, body的质量不被调整. 你必须调用ResetMassData方法来促使动画改变.

fixture->SetDensity(5.0f);

body->ResetMassData();

### Friction

Friction is used to make objects slide along each other realistically. Box2D supports static and dynamic friction, but uses the same parameter for both. Friction is simulated accurately in Box2D and the friction strength is proportional to the normal force (this is called Coulomb friction). The friction parameter is usually set between 0 and 1, but can be any non-negative value. A friction value of 0 turns off friction and a value of 1 makes the friction strong. When the friction force is computed between two shapes, Box2D must combine the friction parameters of the two parent fixtures. This is done with the geometric mean:

摩擦力用于造成对象彼此真实的滑动. Box2D支持静态和动态摩擦力, 但是可以对两个使用相同的参数. 摩擦力在Box2D中精确的模拟, 同时摩擦力度是与法向力(叫做库伦摩擦力)成比例的.虽然摩擦力参数通常被设置为0到1之间, 但是能够被设置为非负的任何值. 0值的摩擦力关闭了摩擦, 同时1值的摩擦力非常强. 当摩擦力是计算于两个shape之间时, Box2D必须组合两个fixture的摩擦力参数.以下是所做的几何意义:

float32 friction;

friction = sqrtf(shape1->friction \* shape2->friction);

So if one fixture has zero friction then the contact will have zero friction.

所以如果一个fixure的摩擦力是0, 那么与之接触将拥有0的摩擦力.

### Restitution

Restitution is used to make objects bounce. The restitution value is usually set to be between 0 and 1. Consider dropping a ball on a table. A value of zero means the ball won't bounce. This is called an inelastic collision. A value of one means the ball's velocity will be exactly reflected. This is called a perfectly elastic collision. Restitution is combined using the following formula.

回弹恢复(Restitution)用于使对象反弹. 回弹值通常设置在0和1之间.想象一个球落到桌子上. 0值意味着球将不会反弹.叫做无弹性碰撞. 值为1意味着球的速度将是精确的反弹(速度不减). 叫做完美弹性碰撞. 使用以下公式计算弹性恢复的组合值.

float32 restitution;

restitution = b2Max(shape1->restitution, shape2->restitution);

Fixtures carry collision filtering information to let you prevent collisions between certain game objects.

Fixtures携带碰撞筛选信息, 以让你防止确定的游戏对象碰撞.

When a shape develops multiple contacts, restitution is simulated approximately. This is because Box2D uses an iterative solver. Box2D also uses inelastic collisions when the collision velocity is small. This is done to prevent jitter.

当一个shape显露多个接触, 回弹恢复系数被近似的模拟. 这是因为Box2D使用一个迭代程序. Box2D也使用无弹性碰撞, 当碰撞速度很小的时候. 这样做是为了防止抖动.

### Filtering筛选

Collision filtering allows you to prevent collision between fixtures. For example, say you make a character that rides a bicycle. You want the bicycle to collide with the terrain and the character to collide with the terrain, but you don't want the character to collide with the bicycle (because they must overlap). Box2D supports such collision filtering using categories and groups.

碰撞筛选允许你防止fixture之间的碰撞. 例如, 说你创造一个角色骑在自行车上. 你想要自行车与地形之间的碰撞和角色与地形之间的碰撞, 但是你不需要角色和自行车之间的碰撞(因为他们之间必须重叠).Box2D使用分类和分组支持这样的碰撞筛选.

Box2D supports 16 collision categories. For each fixture you can specify which category it belongs to. You also specify what other categories this fixture can collide with. For example, you could specify in a multiplayer game that all players don't collide with each other and monsters don't collide with each other, but players and monsters should collide. This is done with masking bits. For example:

Box2D支持16种碰撞分类. 你能指定每一个fixture的分类所属. 你也能够指定这个fixture能够和其他的分类碰撞.例如, 你能够指定游戏中的多个玩家彼此之间不会相互碰撞, 怪物之间也不会彼此碰撞, 但是玩家和怪物会发生碰撞. 这使用遮罩的位操作来完成. 如下:

playerFixtureDef.filter.categoryBits = 0x0002;

monsterFixtureDef.filter.categoryBits = 0x0004;

playerFixtureDef.filter.maskBits = 0x0004;

monsterFixtureDef.filter.maskBits = 0x0002;

Here is the rule for a collision to occur:

下面是碰撞产生的规则:

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

}

Collision groups let you specify an integral group index. You can have all fixtures with the same group index always collide (positive index) or never collide (negative index). Group indices are usually used for things that are somehow related, like the parts of a bicycle. In the following example, fixture1 and fixture2 always collide, but fixture3 and fixture4 never collide.

碰撞组让你指定一个整型的组索引.你能够携带全部的fixtures使用相同的组索引, 正索引总是碰撞, 负索引从不碰撞.组索引通常用于的地方是与某种方法有关联, 就像自行车的部分. 在下面的例子中, fixture1和fixture2总是碰撞的, 但是fixture3和fixture4从不碰撞.

fixture1Def.filter.groupIndex = 2;

fixture2Def.filter.groupIndex = 2;

fixture3Def.filter.groupIndex = -8;

fixture4Def.filter.groupIndex = -8;

Collisions between fixtures of different group indices are filtered according the category and mask bits. In other words, group filtering has higher precedence than category filtering.

根据类别和遮罩的位操作, 碰撞在不同组索引的fixtures之间被过滤. 换句话说, 组过滤的优先级高于类别过滤的优先级.

即:

1. 在组索引相同的情况下, 如果组索引是正数且相同, 那么必定会有碰撞检测, 无论category 和 mask 参数是否满足过滤条件; 如果组索引是负数且相同, 那么必定不会有碰撞检测, 无论category 和 mask 参数是否满足过滤条件;

2. 在组索引不相同的情况下, 无论组索引是什么数, 都会再判断category 和 mask;

3. 组索引为 0 时, 不属于任何组, 都会去判断category 和 mask.

Note that additional collision filtering occurs in Box2D. Here is a list:

注意: 在Box2D中额外的碰撞筛选, 如下所示:

* A fixture on a static body can only collide with a dynamic body.
* A fixture on a kinematic body can only collide with a dynamic body.
* Fixtures on the same body never collide with each other.
* You can optionally enable/disable collision between fixtures on bodies connected by a joint.
* 在静态body上的fixture只能和动态body相碰撞.
* 在一个运动学body上的fixture只能和动态body相碰撞.
* 在相同body上的fixture将永远不会彼此碰撞.
* 通过一个joint连接bodies,而你能够随意启用/禁用在这些body上的fixtures之间的碰撞.

Sometimes you might need to change collision filtering after a fixture has already been created. You can get and set the b2Filter structure on an existing fixture using b2Fixture::GetFilterData and b2Fixture::SetFilterData. Note that changing the filter data will not add or remove contacts until the next time step (see the World class).

某些时候你可能需要在一个fixture已经被建立之后改变碰撞筛选. 你能够使用b2Fixture::GetFilterData和b2Fixture::SetFilterData函数在一个已存在的fixture上get和set b2Filter结构.注意, 改变的filter数据将不会被添加或者移除接触, 直到下一个时间步为止(查看World类).

## Sensors感应器

Sometimes game logic needs to know when two fixtures overlap yet there should be no collision response. This is done by using sensors. A sensor is a fixture that detects collision but does not produce a response.

某些时候游戏逻辑需要知道两个fixture是否重叠, 但是不应该有碰撞响应. 这通过使用感应器完成. 感应器是一个fixture, 它检测碰撞,但是不产生响应.

You can flag any fixture as being a sensor. Sensors may be static or dynamic. Remember that you may have multiple fixtures per body and you can have any mix of sensors and solid fixtures.

你能够标记任何一个fixture作为感应器. 感应器可能是静态或者动态的. 记得你可能会拥有多个fixture在每个body上, 并且你能任意混合感应器和固体fixture.

Sensors do not generate contact points. There are two ways to get the state of a sensor:

感应器不会产生接触点. 有两个方式获得感应器的状态:

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

# Bodies

## About

Bodies have position and velocity. You can apply forces, torques, and impulses to bodies. Bodies can be static, kinematic, or dynamic. Here are the body type definitions:

Bodies拥有位置和速度. 你能应用力, 扭矩, 和冲量到bodies上. Bodies能是静态的, 运动学的, 或者是动态的. 这里有一些body类型的定义:

#### b2\_staticBody

A static body does not move under simulation and behaves as if it has infinite mass. Internally, Box2D stores zero for the mass and the inverse mass. Static bodies can be moved manually by the user. A static body has zero velocity. Static bodies do not collide with other static or kinematic bodies.

静态body, 在模拟环境下不会移动, 而且就像有无限的质量. 在内部, Box2D存储的质量和逆质量的值为0.静态bodies能通过用户手动移动. 一个静态body有0的速度. 静态bodies不会和静态或者运动学bodies碰撞.

#### b2\_kinematicBody

A kinematic body moves under simulation according to its velocity. Kinematic bodies do not respond to forces. They can be moved manually by the user, but normally a kinematic body is moved by setting its velocity. A kinematic body behaves as if it has infinite mass, however, Box2D stores zero for the mass and the inverse mass. Kinematic bodies do not collide with other static or kinematic bodies.

运动学body依照它的速度在模拟情况下移动. 运动学bodies不会响应力. 他们能够通过用户手动移动, 但是通常一个运动学body通过设置速度来移动. 运动学body表现的就像有无限的质量, 不管怎样, Box2D存储质量和逆质量为0. 运动学bodies不会和其他的静态或者运动学bodies相碰撞.

#### b2\_dynamicBody

A dynamic body is fully simulated. They can be moved manually by the user, but normally they move according to forces. A dynamic body can collide with all body types. A dynamic body always has finite, non-zero mass. If you try to set the mass of a dynamic body to zero, it will automatically acquire a mass of one ,.

动态body被完全的模拟. 他们能够通过用户手动移动, 但是通常他们根据力来移动. 动态body能够和所有的body类型相碰撞. 动态body总是有有限的, 非0的质量. 如果试图设置动态body的质量为0, 那么它将自动获得1KG的质量.

Bodies are the backbone for fixtures. Bodies carry fixtures and move them around in the world. Bodies are always rigid bodies in Box2D. That means that two fixtures attached to the same rigid body never move relative to each other.

对于fixture, body是支柱. Bodies携带fixtures并且在世界中移动. Bodies在Box2D中总是钢体. 这意味着两个fixture依附相同的钢体, 但是之间永远不会产生相对移动.

Fixtures have collision geometry and density. Normally, bodies acquire their mass properties from the fixtures. However, you can override the mass properties after a body is constructed. This is discussed below.

Fixtures有碰撞集合图形和密度. 通常情况下, bodies需要从fixture那里知道他们的质量. 不管怎样, 你能够在body被构建后重写他们的质量属性. 这将在后面讨论.

You usually keep pointers to all the bodies you create. This way you can query the body positions to update the positions of your graphical entities. You should also keep body pointers so you can destroy them when you are done with them.

你通常需要保留所有创建的body对象指针. 这个方式能够帮助你查询body的位置, 用来更新你的图形实体. 你同样应该保留body指针, 因为当你完成他们的使用时能够销毁他们.

## Body Definition

Before a body is created you must create a body definition (b2BodyDef). The body definition holds the data needed to create and initialize a body.

在Body建立之前, 你必须建立一个body定义(b2BodyDef). Body定义拥有的数据被需要于建立和初始化一个body.

Box2D copies the data out of the body definition; it does not keep a pointer to the body definition. This means you can recycle a body definition to create multiple bodies.

Box2D从body定义中复制出数据; 它不保留body定义的指针. 这意味这你能够重用一个body定义建立多个bodies.

Let’s go over some of the key members of the body definition.

来看看body定义的主要成员吧, 圣光与你同在.

### Body Type

As discussed at the beginning of this chapter, there are three different body types: static, kinematic, and dynamic. You should establish the body type at creation because changing the body type later is expensive.

作为这一章讨论的开端, 介绍了3个不同的body类型: 静态, 运动学和动态. 你应该在创建时设置body类型, 因为这之后改变body类型的开销会很大.

bodyDef.type = b2\_dynamicBody;

Setting the body type is mandatory.

设置body的类型是强制性的.

### Position and Angle

The body definition gives you the chance to initialize the position of the body on creation. This has far better performance than creating the body at the world origin and then moving the body.

在创建body的时候, body定义会给你初始化位置的机会. 这样的执行效率会比在世界的原始点创建body, 然后再移动body的操作好很多.

**Caution**

Do not create a body at the origin and then move it. If you create several bodies at the origin, then performance will suffer.

**小心:**

不要在原点创建body然后再移动他. 如果你创建若干个body在原点, 那么会降低执行效率.

A body has two main points of interest. The first point is the body's origin. Fixtures and joints are attached relative to the body's origin. The second point of interest is the center of mass. The center of mass is determined from mass distribution of the attached shapes or is explicitly set with b2MassData. Much of Box2D's internal computations use the center of mass position. For example b2Body stores the linear velocity for the center of mass.

body有两个主要的兴趣点(points of interest). 第一个点是body的原点. Fixture和joint都相对于body的原点被依附上. 第二个点是重心(center of mass).重心通过附加的shapes分配或者通过使用b2MassData明确的设置来决定的. Box2D的很多内部计算是使用重心位置的. 例如b2Body是对于重心来存储线性速度的.

When you are building the body definition, you may not know where the center of mass is located. Therefore you specify the position of the body's origin. You may also specify the body's angle in radians, which is not affected by the position of the center of mass. If you later change the mass properties of the body, then the center of mass may move on the body, but the origin position does not change and the attached shapes and joints do not move.

当你创建body定义的时候, 你可能不知道重心被定位在哪. 因此你指定body原点的位置. 你也可能以弧度为单位指定body的角度, 这些做法都不会影响重心的位置. 如果你之后改变了body的质量属性, 那么重心可能在body上移动, 但是原点的位置不会改变, 同时依附的shapes和joints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.

### Damping

Damping is used to reduce the world velocity of bodies. Damping is different than friction because friction only occurs with contact. Damping is not a replacement for friction and the two effects should be used together.

阻尼(Damping)用于减少body的世界速度. 阻尼和摩擦力不同, 因为摩擦力只发生在接触的时候. 阻尼不能替换摩擦力, 并且两个效果能同时使用.

Damping parameters should be between 0 and infinity, with 0 meaning no damping, and infinity meaning full damping. Normally you will use a damping value between 0 and 0.1. I generally do not use linear damping because it makes bodies look floaty.

阻尼参数应该在0和无穷大之间, 0表示没有阻尼, 无穷大意味着满阻尼. 通常你将使用阻尼的值在0和0.1之间. 我通常不使用线性阻尼, 因为他让body看起来很飘飘然.

bodyDef.linearDamping = 0.0f;

bodyDef.angularDamping = 0.01f;

Damping is approximated for stability and performance. At small damping values the damping effect is mostly independent of the time step. At larger damping values, the damping effect will vary with the time step. This is not an issue if you use a fixed time step (recommended).

阻尼类似稳定性和性能. 小阻尼值的阻尼效果几乎不受时间步的约束. 大阻尼值的阻尼效果将随着时间步变化. 如果你使用固定时间步的话,(推荐), 这将不会是问题.

### Gravity Scale

You can use the gravity scale to adjust the gravity on a single body. Be careful though, increased gravity can decrease stability.

你能使用重力缩放(gravity scale)调整单个body的重力. 不过要小心, 增加重力能减少稳定性.

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

bodyDef.gravityScale = 0.0f;

### Sleep Parameters

What does sleep mean? Well it is expensive to simulate bodies, so the less we have to simulate the better. When a body comes to rest we would like to stop simulating it.

slepp是什么意思呢?模拟物体的成本是很昂贵的, 所以减少一些物体, 模拟的效果会变好. 当一个body停止移动, 我们将停止模拟它.

When Box2D determines that a body (or group of bodies) has come to rest, the body enters a sleep state which has very little CPU overhead. If a body is awake and collides with a sleeping body, then the sleeping body wakes up. Bodies will also wake up if a joint or contact attached to them is destroyed. You can also wake a body manually.

当Box2D决定一个body(或者一组bodies)停止移动的时候, body进入sleep状态, 这时只使用一点点CPU.如果一个body是醒着(awake)的, 那么它将与sleep的body相碰撞, 那时sleep的body将被唤醒. 如果一个joint或者接触(contact)被摧毁, 那么body也将会被唤醒. 你也能够手动唤醒body.

The body definition lets you specify whether a body can sleep and whether a body is created sleeping.

body定义让你指定一个body是否能够sleep和是否被创建为sleep状态.

bodyDef.allowSleep = true;

bodyDef.awake = true;

### Fixed Rotation

You may want a rigid body, such as a character, to have a fixed rotation. Such a body should not rotate, even under load. You can use the fixed rotation setting to achieve this:

你可能想要一个钢体, 比如一个角色, 固定旋转. 即使在负载情况下, 这样的body不应该旋转. 你能使用固定旋转设置达到这样的目的.

bodyDef.fixedRotation = true;

The fixed rotation flag causes the rotational inertia and its inverse to be set to zero.

固定旋转标识引起转动惯量和逆惯量被设置为0.

### Bullets

Game simulation usually generates a sequence of images that are played at some frame rate. This is called discrete simulation. In discrete simulation, rigid bodies can move by a large amount in one time step. If a physics engine doesn't account for the large motion, you may see some objects incorrectly pass through each other. This effect is called tunneling.

游戏模拟通常生成一系列图像, 这些图像以某一帧率播放. 这叫做离散模拟.在离散模拟中, 钢体能够在一个时间步大量的移动. 如果一个物理引擎没有对大范围移动负责, 那么你可能看到一些对象错误的彼此穿透. 这个现象叫做隧道效应(tunneling).

By default, Box2D uses continuous collision detection (CCD) to prevent dynamic bodies from tunneling through static bodies. This is done by sweeping shapes from their old position to their new positions. The engine looks for new collisions during the sweep and computes the time of impact (TOI) for these collisions. Bodies are moved to their first TOI and then halted for the remainder of the time step.

默认情况下, Box2D使用连续的碰撞检测(continuous collision detection: CCD)以防止动态body穿透静态body.这通过扫描形状从他们的之前位置到他们的新位置来完成.引擎在扫描期间会搜索新的碰撞, 并且为这些碰撞计算碰撞时间(TOI). Body被移动到他们首次发生碰撞的时间, 然后暂停剩余的时间步.

Normally CCD is not used between dynamic bodies. This is done to keep performance reasonable. In some game scenarios you need dynamic bodies to use CCD. For example, you may want to shoot a high speed bullet at a stack of dynamic bricks. Without CCD, the bullet might tunnel through the bricks.

通常CCD不在两个动态的body之间使用. 这样做是保持执行效率的合理性. 在一些游戏场景中, 你需要动态body使用CCD. 例如, 你可能想要对一堆动态砖块发射一枚高速的子弹. 如果没有CCD, 子弹可能会穿透砖块.

Fast moving objects in Box2D can be labeled as bullets. Bullets will perform CCD with both static and dynamic bodies. You should decide what bodies should be bullets based on your game design. If you decide a body should be treated as a bullet, use the following setting.

在Box2D中快速移动的物体被标记为子弹. 子弹将在静态和动态body之间执行CCD. 你应该在你的游戏设计中决定哪些body应该被标记为子弹. 如果你决定一个body应该被当作子弹来看待, 使用以下的设置.

bodyDef.bullet = true;

The bullet flag only affects dynamic bodies.

bullet标识只影响动态body.

Box2D performs continuous collision sequentially, so bullets may miss fast moving bodies.

Box2D顺序的执行连续碰撞, 所以子弹可能会错过快速移动的body.

### Activation

You may wish a body to be created but not participate in collision or dynamics. This state is similar to sleeping except the body will not be woken by other bodies and the body's fixtures will not be placed in the broad-phase. This means the body will not participate in collisions, ray casts, etc.

你可能希望一个被建立的body, 但是不参与碰撞或者动态的. 这个状态和sleep相同, 除了这个body不会被另外的body要求工作, 同时body的fixture将不会放置在broad-phase阶段. 这意味这body将不会参与碰撞, 射线投影, 等.

You can create a body in an inactive state and later re-activate it.

你能够创建一个body以一个不激活状态, 并且之后重新激活它.

bodyDef.active = true;

Joints may be connected to inactive bodies. These joints will not be simulated. You should be careful when you activate a body that its joints are not distorted.

Joints可能连接到不活跃的body. 这些joint将不会被模拟. 你应该小心, 当你激活一个body时, 这些joint不会扭曲.

### User Data

User data is a void pointer. This gives you a hook to link your application objects to bodies. You should be consistent to use the same object type for all body user data.

用户数据(userData)是一个空指针. 这提供给你一个钩子链接你的应用程序对象到body. 你应该始终如一对全部body的userData使用相同的对象类型.

b2BodyDef bodyDef;

bodyDef.userData = &myActor;

## Body Factory

Bodies are created and destroyed using a body factory provided by the world class. This lets the world create the body with an efficient allocator and add the body to the world data structure.

世界类提供body的工厂方法以创建和销毁body对象. 这让世界有效率的分配body内存和添加body到世界数据结构中.

Bodies can be dynamic or static depending on the mass properties. Both body types use the same creation and destruction methods.

Body能够是动态或者静态的, 取决于质量属性. 两种body类型使用相同的创建和销毁方法.

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

... do stuff ...

myWorld->DestroyBody(dynamicBody);

dynamicBody = NULL;

**Caution**

You should never use new or malloc to create a body. The world won't know about the body and the body won't be properly initialized.

注意:

你永远不应该使用new或者malloc方法创建一个body. 世界对象不会知道这样body存在, 同时body将不会被正确的初始化.

Static bodies do not move under the influence of other bodies. You may manually move static bodies, but you should be careful so that you don't squash dynamic bodies between two or more static bodies. Friction will not work correctly if you move a static body. Static bodies never collide with static or kinematic bodies. It is faster to attach several shapes to a static body than to create several static bodies with a single shape on each one. Internally, Box2D sets the mass and inverse mass of static bodies to zero. This makes the math work out so that most algorithms don't need to treat static bodies as a special case.

静态body不会在其他body影响下而移动.你可以手动移动静态body, 但是你应该小心, 以便你不会在两个或者更多的静态body之间挤压动态body.如果你移动一个静态body摩擦力将不会正确的工作. 静态body永远不会与静态或者运动学body发生碰撞. 依附若干个shapes到一个静态物体比分别使用一个shape创建若干个静态body来得快.在内部, Box2D设置静态body的质量和逆质量为0. 这使得大多数数学算法的实现不会把静态body当作特殊情况来对待.

Box2D does not keep a reference to the body definition or any of the data it holds (except user data pointers). So you can create temporary body definitions and reuse the same body definitions.

Box2D不会保留body定义的引用或者任何一个它持有的数据(除了userData指针). 所以你能够创建临时body定义, 并且重用相同的body定义.

Box2D allows you to avoid destroying bodies by deleting your b2World object, which does all the cleanup work for you. However, you should be mindful to nullify body pointers that you keep in your game engine.

Box2D允许你通过删除你的b2World对象来避开body的销毁, 它会为你做所有的清除工作. 无论如何, 你应该谨慎的把保留在游戏引擎中的body指针作废.

When you destroy a body, the attached fixtures and joints are automatically destroyed. This has important implications for how you manage shape and joint pointers.

当你销毁一个body, 依附的fixture和joint会自动的被销毁.这对于你如何管理shape和joint指针有很重要的含义.

## Using a Body

After creating a body, there are many operations you can perform on the body. These include setting mass properties, accessing position and velocity, applying forces, and transforming points and vectors.

在创建一个body之后, 可能会有很多的操作执行在body上. 这些包括设置质量属性, 访问位置和速度, 应用力, 和转变点和向量.

### Mass Data

Every body has a mass (scalar), center of mass (2-vector), and rotational inertia (scalar). For static bodies, the mass and rotational inertia are set to zero. When a body has fixed rotation, its rotational inertia is zero.

每一个body有一个质量(标量), 重心(二维向量), 和旋转惯量(标量). 对于静态body, 质量和旋转惯量都是0. 当一个body有了固定旋转, 它的旋转惯量是0.

Normally the mass properties of a body are established automatically when fixtures are added to the body. You can also adjust the mass of a body at run-time. This is usually done when you have special game scenarios that require altering the mass.

通常情况下body的质量属性是在fixture被添加到body的时候自动确定的. 你也能够在运行时调整body的质量属性. 这通常用在你有一个特殊游戏场景需要更改质量.

void SetMassData(const b2MassData\* data);

After setting a body's mass directly, you may wish to revert to the natural mass dictated by the fixtures. You can do this with:

当你直接的设置完body的质量后, 你可能希望恢复到通过fixtures指定的自然质量. 你可以调用这个:

void ResetMassData();

The body's mass data is available through the following functions:

可以通过以下的函数获得有效的body质量.

float32 GetMass() const;

float32 GetInertia() const;

const b2Vec2& GetLocalCenter() const;

void GetMassData(b2MassData\* data) const;

### State Information

There are many aspects to the body's state. You can access this state data efficiently through the following functions:

Body的状态有很多面貌. 你能够高效的通过下面的函数访问他们:

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;

### Position and Velocity

You can access the position and rotation of a body. This is common when rendering your associated game actor. You can also set the position, although this is less common since you will normally use Box2D to simulate movement.

你能够访问body的位置和旋转角度. 这在渲染你关联的游戏演员时是很常见的. 你也能设置位置, 虽然这不常见, 因为通常使用Box2D模拟移动.

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

const b2Transform& GetTransform() const;

const b2Vec2& GetPosition() const;

float32 GetAngle() const;

You can access the center of mass position in local and world coordinates. Much of the internal simulation in Box2D uses the center of mass. However, you should normally not need to access it. Instead you will usually work with the body transform. For example, you may have a body that is square. The body origin might be a corner of the square, while the center of mass is located at the center of the square.

你能够在局部和世界坐标系中访问重心的位置.在Box2D中的很多内部模拟使用重心. 无论如何, 你通常不需要访问他. 反而你会经常与body的变形一同工作. 例如, 你可以有一个正方形的body.Body原点可能是正方形的一个角, 然而重心的局部坐标在正方形的中心.

const b2Vec2& GetWorldCenter() const;

const b2Vec2& GetLocalCenter() const;

You can access the linear and angular velocity. The linear velocity is for the center of mass. Therefore, the linear velocity may change if the mass properties change.

你能够访问线速度和角速度. 线速度鉴于重心. 因此线速度可能随质量属性而改变.

# Joints

## About

Joints are used to constrain bodies to the world or to each other. Typical examples in games include ragdolls, teeters, and pulleys. Joints can be combined in many different ways to create interesting motions.

Joints用于在世界中束缚body或者彼此束缚. 游戏中典型的示例: 布偶, 跷跷板, 滑轮. joints能够以很多不同的方式组合出有趣的运动.

Some joints provide limits so you can control the range of motion. Some joint provide motors which can be used to drive the joint at a prescribed speed until a prescribed force/torque is exceeded.

一些joints提供限制, 所以你能够控制运动的范围. 一些joint提供马达(motor), 用一个规定的速度驱动joint, 直到超过规定的力/扭力(force/torque)才停止.

Joint motors can be used in many ways. You can use motors to control position by specifying a joint velocity that is proportional to the difference between the actual and desired position. You can also use motors to simulate joint friction: set the joint velocity to zero and provide a small, but significant maximum motor force/torque. Then the motor will attempt to keep the joint from moving until the load becomes too strong.

Joints马达能够以很多方式被使用. 你能通过指定joint的速度使马达控制位置, 而速度是与起始点和目的点之间的差值成比例的. 你也能使用马达模拟joint的摩擦: 设置joint速度为0并且提供一个小的, 但有效的最大马达力/扭力(force/torque). 那时马达将企图保持joint移动, 直到负载变为最大值.

## The Joint Definition

Each joint type has a definition that derives from b2JointDef. All joints are connected between two different bodies. One body may static. Joints between static and/or kinematic bodies are allowed, but have no effect and use some processing time.

每一个joint类型有一个定义, 起源于b2JointDef. 全部的joints连接于两个不同的body之间. 其中一个body可能是静态的. Joints在静态和/或运动学body之间是允许的, 但是不会有任何效果, 并且开销一些处理时间.

You can specify user data for any joint type and you can provide a flag to prevent the attached bodies from colliding with each other. This is actually the default behavior and you must set the collideConnected Boolean to allow collision between to connected bodies.

你能为任何一个joint类型指定userData, 并且提供一个标识以防止依附的body彼此相互碰撞. 这些实际上是默认行为, 但是你必须设置collisionConnected布尔值以允许两个body之间的碰撞.

Many joint definitions require that you provide some geometric data. Often a joint will be defined by anchor points. These are points fixed in the attached bodies. Box2D requires these points to be specified in local coordinates. This way the joint can be specified even when the current body transforms violate the joint constraint --- a common occurrence when a game is saved and reloaded. Additionally, some joint definitions need to know the default relative angle between the bodies. This is necessary to constrain rotation correctly.

很多joint定义需要你提供一些几何数据. 通常一个joint将通过定位点定义.这些点固定在依附的body上. Box2D需要这些点以局部坐标指定.这个方式的 joint 能够被详细的说明, 即使当当前body变形违反joint约束的时候----通常情况下发生在当游戏存储和读取的时候. 此外, 一些joint定义需要知道body之间的默认相对角度. 这有必要正确的约束旋转.

Initializing the geometric data can be tedious, so many joints have initialization functions that use the current body transforms to remove much of the work. However, these initialization functions should usually only be used for prototyping. Production code should define the geometry directly. This will make joint behavior more robust.

初始化几何数据是非常乏味的, 所以很多joint拥有初始化函数通过使用当前body的变形移除了大多数的工作. 不管如何, 这些初始化函数通常应该只被用于原型设计. 生产代码应该直接定义几何图形. 这将会使joint的行为更加健壮.

The rest of the joint definition data depends on the joint type. We cover these now.

剩余的joint定义数据依赖于joint的类型. 我们在下面涉及.

## Joint Factory

Joints are created and destroyed using the world factory methods. This brings up an old issue:

Joints使用世界的工厂方法来建立和销毁. 这提出一个老问题:

**Caution**

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.

注意:

不要尝试创建一个joint在stack上或者在heap上使用new 或者malloc.你必须使用b2World类的create和destroy方法创建和销毁body和joint.

Here's an example of the lifetime of a revolute joint:

这里有一个旋转joint的生命期示例:

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;

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.

在他们被销毁后使你的指针作废, 这总是一个很好的习惯. 这将使程序在控制管理阶段崩溃, 如果你尝试重用指针的话.

The lifetime of a joint is not simple. Heed this warning well:

Joint的生命周期不简单. 留意这个警告:

**Caution**

Joints are destroyed when an attached body is destroyed.

注意:

当joint依附的body被销毁时, 该joint也被销毁.

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.

这个预防措施并不总是有必要的. 你可能组织你的游戏引擎以便joints总是销毁在依附的body之前. 在这种情况下你不需要实现侦听器(listener)类. 更多细节请查看(Implicit Destruction)这节.

## 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.

很多模拟建立joint然后不再访问他们, 直到他们被销毁. 不管怎样, 有很多有用的数据包含于joint, 让你能够使用以建立一个丰富的模拟.

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

首先, 你能从joint获取body, 定位点, 和用户数据.

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.

全部的joint有一个反作用力和扭矩. 这个反作用力应用于body2的定位点. 你能使用反作用力削弱joint或者触发其他游戏事件.这些函数可能会做一些计算指令, 所以如果你不需要结果就不要使用他们.

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.

最简单的joint就是距离关节, 距离的意思就是分别两个body上的点的距离是一个常量. 当你指定一个距离关节时, 两个body应该已经在适当的地方. 然后你以世界坐标系指定两个定位点. 第一个定位点连接body1, 第二个定位点连接body2. 这些点暗示着距离长度的常量.



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

这里是一个距离关节定义的示例. 在这种情况下我们决定允许body能够碰撞.

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.

距离关节也能被做成柔软的, 就像一个橡皮筋连接. 查看testbed中的Web例子, 就知道它有什么样子的行为了.

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.

调整两个定义中的常数取得柔和的效果: 频率和阻尼系数. 想像频率如同一个谐波振荡器的频率(像吉他弦). 频率的指定以赫兹为单位. 典型的频率应该小于一半的时间步频率. 所以如果你使用一个60Hz的时间步, 距离关节的频率应该小于30Hz. 原因是与奈奎斯特频率(Nyquist frequency)有关.

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).

阻尼系数是无因次的, 并且是在0和1之间的, 但可以更大. 为1时, 阻尼是临界的(所有振幅应该消失).

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.

一个转动关节推动两个body共享一个普通的定位点, 通常叫做枢纽点. 旋转关节有一个单独的自由角度: 与两个body的旋转相关. 叫做关节角度.



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.

指定一个转动关节你需要在世界坐标系提供两个body和一个单独定位点. 初始化函数假定bodies已经在正确的位置上.

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

在这个例子中, 两个body通过一个转动关节连接, 交接点是第一个body的重心.

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.

转动关节角度是正的, 当bodyB对于角度点逆时针旋转时. 像所有在Box2D中的角度, 旋转角度是以弧度为标准的. 当使用initialize()创建joint时, 约定旋转关节角度为0, 不管当前两个body的旋转角度.

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.

在一些情况下你可能希望控制关节的角度. 为此, 旋转关节能够随意的模拟一个关节的界限和/或一个马达.

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.

一个joint界限促使关节角度保留在较低的和较高的范围之间. 这个界限将需要尽可能多的扭矩促使他发生. 界限的范围应该包含0, 否则当模拟开始时joint就会倾斜.

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:

Joint马达允许你指定joint的速度(角的时间导数). 速度可以是负数或正数. 马达能够有无穷大的力, 但是这通常不令人满意.回想起一个永恒的话题:

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

当一个不可抗拒的力量遇到了一个不可移动的物体, 会发生什么?

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.

我可以告诉你它不可爱. 所以你可以提供一个最大限度的扭力给joint的马达. Joint马达将维持指定的速度除非需要的扭力超过指定的最大限度.当最大限度扭力被超过, joint将慢下来, 甚至反转. 这里可以这样理解, Revolute joint 要旋转就必须要有 Torque 的值, 如果该值能够促使 bodyB 旋转, 那么 bodyB 将以设置的 Speed 为角速度旋转; 但是这里还有一个制约, 如果 bodyB 的受力超过最大的力矩设置中的作用力, 那么这时 bodyB 还是会根据受力情况响应物理效果, 而不是继续按设置的角速度旋转.

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.

你能使用joint马达模拟joint摩擦. 只是设置joint速度为0, 同时设置最大限度扭力为一些小的值, 但是是有效的值. 马达将试图预防joint旋转, 但是将屈服有效的负载.

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.

这里是一个上文的旋转joint定义修订版; 这次joint有极限值和一个激活马达. 马达是模拟joint的机构.

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.

你能访问一个旋转joint的角度, 速度和马达扭力.

float32 GetJointAngle() const;

float32 GetJointSpeed() const;

float32 GetMotorTorque() const;

You also update the motor parameters each step.

你也可以每一步更新马达参数.

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.

Joint马达有一些有趣的能力. 每个时间步你能更新joint的速度, 所以你能使joint来回移动, 就像正弦波, 或者依照你想要的任何函数.

... 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:

你也能使用joint马达跟踪一个期望的joint角度. 例如:

... 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.

通常你的gain参数不应该太大. 否则你的joint可能变为不稳定的.

## Prismatic Joint移动关节

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.

移动关节允许的两个body沿着指定轴做相对平移.移动关节防止有相对的旋转.因此, 移动关节有一个单独的自由角度.



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:

移动关节定义相似于旋转关节的描述; 只是平移与角度替换和力与扭矩替换. 按照这样推理, 使用一个关节界限和一个摩擦马达提供一个移动关节的定义示例:

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.

旋转关节有一个隐式的轴从屏幕伸出来. 移动关节需要一个明确的轴平行于屏幕. 这个轴固定两个body, 并且遵循他们的运动.

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.

像旋转关节, 移动关节当初始化建立的时候平移值是0.所以确定0是在你的最低和最高平移界限之间.

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

使用移动关节与使用旋转关节相似. 这里是有关的成员函数:

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.

滑轮用于创建一个理想化的滑轮. 滑轮连接两个body到地面, 并且两个body彼此相连.当一个body向上时, 另外一个向下. 滑轮绳的总长度依照初始化配置恒定.

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.

你能提供一个系数模拟砖头和滑车. 这会导致滑轮的一边延伸的比另外一边更快. 在相同的时间约束力一边小于另一边. 你能够使用这个创建机械杠杆.

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.

例如, 如果系数是2, 那么length1的变化将是length2的两倍. 另外依附于body1的绳子力将会是依附于body2绳子力的一半.



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.

当一边充分延伸的时候, 滑轮会变得很麻烦. 在另外一边的绳子将只有0的长度. 此时此刻约束方程式变得单一(不好). 你应该设置碰撞shape, 以防止这种情况发生.

Here is an example pulley definition:

这里是滑轮示例的定义:

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.

滑轮关节提供当前的长度.

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.

如果你想创建一个复杂的机械装置, 你可能会想要使用齿轮. 本质上你能在Box2D中使用复合shape塑造齿轮齿创建一个齿轮. 这不是很有效率并且可能使创作人感到很乏味. 同时你必须小心的整理齿轮, 因为牙齿必须光滑的啮合. Box2D有一个简单的方法生成齿轮: 就是齿轮关节(gear joint).



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

齿轮关节只是用来连接旋转和/或移动关节.

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 units of 1/length.

像滑动系数, 你也能指定一个齿轮系数. 不管怎样, 在这种情况下的齿轮系数能够是负数. 同时记住当一个关节是旋转关节(角度)和另外一个关节是移动关节(平移)时, 齿轮系数将有单位长度或者单位的长度倒数.

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.

这是一个齿轮关节的例子. bodyA和bodyB是来自于两个joint中的任何一个body, 只要他们不是相同的body.

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?

注意: 齿轮关节依赖两个其他的joint. 这构成了一个不好的情况. 如果joint被删除会发生什么呢?

**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.

注意:

齿轮关节总应该在旋转/移动关节之前被删除. 否则你的代码将会以一种不好的方式崩溃, 这归应于在齿轮关节中存在独立的指针. 同时你应该在删除任何与body相关的数据之前删除齿轮关节.

## 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.

Mouse joint在testbed中被使用于使用鼠标操作body.它试图驱动在body上的一个点朝当前光标的位置移动. 没有旋转上的限制.

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.

Mouse joint定义有一个目标点, 最大力度, 频率, 和振幅系数. 目标点首先与body的定位点一致. 当多个动态body相互影响时, 最大极限力防止暴力反作用. 你可以设置你喜欢的大小. 频率和振幅系数用以创建一个弹簧/阻尼器效果, 类似于距离关节.

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.

很多用户尝试为了游戏性会修改鼠标关节. 用户经常想获得精确的位置和瞬间的相应. 鼠标在那样的环境中不会工作良好. 你可以考虑使用运动学body来代替他.

## 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.

车轮关节(wheel joint)限定bodyB上一个点到bodyA上的一条线. 车轮关节也提供一个悬挂弹簧. 查看b2WheelJoint.h和Car.h的细节.



## 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.

焊接关节(weld joint)尝试束缚两个body之间的全部相对运动. 查看testbed中的Cantilever.h, 看看焊接点是怎样的行为.

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.

使用焊接关节定义可损坏的结构是非常诱人的. 然而, Box2D的solver是迭代的, 所以joint有一点软. 因此通过焊接关节连接的body链将会是弯曲的.

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.

代替他最好的方法是用多个fixture创建一个可损坏的单个body. 当body被损坏时, 你能够销毁一个fixture, 并且重建一个新的body. 在testbed中查看Breakable示例.

## 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.

绳子关节限制两个点之间的最大距离. 这会预防body链的拉伸, 甚至在高负载的情况下. 查看b2RopeJoint.h和RopeJoint.h的细节.

## 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.

摩擦关节被用于自顶向下的摩擦力. Joint提供2D平移摩擦力和角度摩擦力. 查看b2FrictionJoint.h和ApplyForce.h的细节.

# 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.

Contact是Box2D创建的对象, 用来管理两个fixture之间的碰撞. 如果fixture有孩子, 比如一个shape链, 那么一个contact为每一个关联的孩子存在. 有很多不同种类的contact, 衍生于b2Contact, 用于管理不同种类fixture之间的contact.例如有一个contact类管理多边形与多边形的碰撞, 和另外一个contact类管理圆和圆的碰撞.

Here is some terminology associated with contacts.

这里有一些术语关联到contact.

### contact point接触点

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

接触点是两个shape接触的点. Box2D用少量的点粗略估计接触.

### contact normal接触法向

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

接触法向是一个单位向量, 从一个shape指向另外一个. 根据惯例, 法向是fixtureA指向fixtureB.

### contact separation接触分离

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.

分离是反向渗透(即分开). 当shape重叠的时候, 分离是负的. 它可能在未来的Box2D版本中将使用正分离创建接触点, 所以当一个接触点被报告时, 你可能需要检查符号.

### 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.

在两个凸多边形之间的接触可能产生两个以上的接触点.这两个点使用相同的法向, 所以它们被分成一个contact manifold, 这是一个连续接触区域的近似值.

### 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.

法向力是一个力, 应用于接触点以防止shape的渗透. 为了方便起见, Box2D使用冲量工作. 法向冲量就是法向力乘以时间步.

### tangentimpulse

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

切向力在接触点在被生成, 用以模拟摩擦. 为了方便起见, 作为一个冲量储存.

### contact ids接触标识

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.

Box2D尝试从一个时间步重用接触力的结果, 将其作为初始值推测下一个时间步. Box2D使用接触标识匹配接触点横穿时间步. 标识包含几何图形特征索引,用来帮助区别于其他接触点.

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.

当两个fixture的AABBs重叠时contact被建立. 有时碰撞筛选将防止contact的创建. 随着AABBs停止重叠, contact被销毁.

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.

所以你可能会收集到那些没有接触却被fixture创建的contact(只是他们的AABBs). 好吧, 的确是这样. 这是一个”鸡或蛋”的问题. 我们不知道是否我们需要一个contact对象, 直到一个contact被创建, 并且用其去分析碰撞. 如果shape没有碰撞, 我们能够立刻删除contact, 或者我们只能等待, 直到AABBs停止重叠. Box2D采取了后面的方法, 因为它让系统缓存信息以提高执行效率.

## 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.

如前所述, contact类通过Box2D创建和销毁. Contact对象不能被用户创建. 然而, 你能够访问contact类, 同时与contact对象相互作用.

You can access the raw contact manifold:

你能够访问原始的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.

你有可能修改manifold, 但是这通常不被支持, 并且这是高级用法.

There is a helper function to get the b2WorldManifold:

有一些帮助函数, 以获取b2WorldManifold:

void GetWorldManifold(b2WorldManifold\* worldManifold) const;

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

这使用当前body的位置计算接触点的世界位置.

Sensors do not create manifolds, so for them use:

传感器不创建manifolds, 所以适合它们的使用如下:

bool touching = sensorContact->IsTouching();

This function also works for non-sensors.

这个函数对于非传感器也会工作.

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

你能从一个contact获得fixture. 从这些方法你能得到body.

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.

你能使一个contact无效. 这只是工作在b2ContactListener::PreSolve事件内部, 接下来讨论.

## 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.

你能以若干种方法访问contact. 你能在世界上直接访问contact和body结构. 你也能实现一个contact的侦听器.

You can iterate over all contacts in the world:

你能在世界迭代全部的contact:

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.

你也能迭代一个body上的全部contact. Contact以图的方式存储, 使用了接触边(contact edge)数据结构.

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.

你也能通过接触侦听器访问contact, 在下面讨论.

**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.

注意:

通过b2World和b2Body访问contact可能错过一些发生在时间步中间的短暂contact. 使用b2ContactListener获得大多数精确的结果.

## Contact Listener

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

你能够通过实现b2ContactListener接受contact数据. 接触侦听器支持若干个事件: 开始, 结束, 处理前, 处理后.

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.

注意:

不要保存发送到b2ContactListener的指针. 取而代之, 制造一个深度复制的contact指针数据到你的缓存中. 下面的示例展示了这个方式的做法.

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.

在运行时你能创建一个侦听器示例, 同时使用b2World::SetContactListener注册它.当世界对象存在的时候确定你的侦听器在作用域被保留.

### 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.

当两个fixture开始重叠时被调用. 这个调用适用于传感器和非传感器. 这个事件只能在时间步内部促发.

### 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.

当两个fixture停止重叠时被调用. 这个调用适用于传感器和非传感器. 当一个body销毁时可能会被调用, 所以这个事件能在时间步外被促发.

### 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.

这个调用在碰撞检测之后, 但是在碰撞决议之前. 这里给你一个机会使contact失效, 基于当前配置的情况下. 例如, 你能使用这个回调和调用b2Contact::SetEnabled(false)实现一个单边的平台.Contact凭借碰撞处理, 每次都会被重新启用, 所以你将需要禁用contact在每个时间步. 每个pre—solve事件可能被调用多次每次时间步的每个contact, 这归应于连续的碰撞检测.

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.

Pre-solve事件是确定点的状态和着手处理碰撞速度的好地方.

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.

Post-solve事件在你能够收集碰撞冲量结果的地方. 如果你不关心冲量, 那么你大概只要实现pre-solve事件就行了.

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.

在一个接触回调内改变物理世界以现游戏逻辑是很诱惑人的.例如, 你可能有一个碰撞, 而这个碰撞产生伤害, 并且试图毁灭关联的角色和它的钢体. 然而, Box2D不允许你在一个回调内修改物理世界, 因为你可能毁灭Box2D当前正在处理的对象, 导致孤立的指针.

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.

处理接触点的推荐方法是缓存所有你关心的点, 并且在时间步之后处理它. 你总是应该在时间步之后立即处理接触点; 另外一些其他客户端代码可能会修改物理世界, 使接触缓存无效. 当你处理接触缓存的时候, 你能够修改物理世界, 但是你仍然需要小心, 不要孤立接触点的缓存指针.testbed有处理接触点的例子, 没有孤立指针的危险.

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.

这段代码来自于CollisionProcessing测试, 展示当处理接触缓存的时候如何处理孤立body.这是一段摘录. 确定阅读清单的注释. 这段代码假定所有接触点已经被缓存在b2ContactPoint数组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.

// 我们准备根据接触点销毁一些body. 我们必须缓存应该被销毁的body,

// 因为他们可能属于多个接触点.

const int32 k\_maxNuke = 6;

b2Body\* nuke[k\_maxNuke];

int32 nukeCount = 0;

// Traverse the contact buffer. Destroy bodies that

// are touching heavier bodies.

// 遍历contact缓存. 销毁body, 那些接触的并且更重的body.

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

{

ContactPoint\* point = m\_points + i;

b2Body\* body1 = point->shape1->GetBody();

b2Body\* body2 = point->shape2->GetBody();

float32 mass1 = body1->GetMass();

float32 mass2 = body2->GetMass();

if (mass1 > 0.0f && mass2 > 0.0f)

{

if (mass2 > mass1)

{

nuke[nukeCount++] = body1;

}

else

{

nuke[nukeCount++] = body2;

}

if (nukeCount == k\_maxNuke)

{

break;

}

}

}

// Sort the nuke array to group duplicates.

// 将nuke数组排序, 将重复的指针排在一起, 因为contact起

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

// Destroy the bodies, skipping duplicates.

// 销毁body, 跳过副本

int32 i = 0;

while (i < nukeCount)

{

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

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

{

++i;

}

m\_world->DestroyBody(b);

}

## Contact Filtering接触过滤

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.

时常在游戏中你不希望所有的对象都碰撞. 例如, 你可能想创建一扇门, 只是确信角色能够穿过它. 这叫做接触过滤, 因为一些交互被过滤掉了.

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.

Box2D允许你完成自定义接触过滤, 通过实现一个b2ContactFilter类. 这个类需要你实现一个ShouldCollide函数, 这个函数接受两个b2Shape指针. 你的函数返回true, 如果shapes应该碰撞的话.

The default implementation of ShouldCollide uses the b2Filterdefined in Chapter 6, Fixtures.

ShouldCollide默认的实现是使用了第六章Fixture定义的b2Filter.

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.

在运行时你能创建一个你的接触过滤实例, 并且使用b2World::SetContactFilter注册它. 当世界对象存在的时候, 确保你的过滤对象在作用域中.

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.

b2World类包含body和joint. 它管理模拟的全部方面, 允许异步查询(就像AABB查询和光线投影ray-casts). 你与Box2D的很多交互都将用到b2World对象.

### 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.

创建一个世界相当的简单. 你只需要提供一个重力向量和一个布尔值指示body是否能sleep. 通常情况下你会使用new和delete创建和销毁一个世界.

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.

世界类包含工厂适合于创建和销毁body和joint. 这些工厂已经在body和joing章节讨论过了.有一些与b2World的其他交互我现在会涉及.

### Simulation

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

世界类用于驱动模拟. 你指定一个时间步, 速度和位置迭代次数. 例如:

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.

在时间步之后你能够检查你的body和joint的信息. 很有可能你会取出body的位置, 以便更新你的角色和渲染它们. 你能执行时间步在你游戏循环的任何地方, 但是你应该意识到事物的循序. 例如, 如果你想要在这一帧对新的body获取碰撞结果, 那么你必须在时间步之前创建body.

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.

如同上面HelloWorld教程讨论的, 你应该使用一个固定时间步. 通过使用一个大时间步, 在低帧率场景下你能增加执行效率. 但是通常你应该使用的时间步不大于1/30秒. 一个1/60秒的时间步通常会实现一个高性能的模拟.

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.

迭代次数控制世界中约束解决器(constraint solver)扫描全部接触和joints的次数. 更多的迭代也总是产生更好的模拟. 但是不要用小时间步交换一个大迭代次数. 60Hz和10次迭代是比30Hz和20次迭代好得多.

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.

在时间步之后, 你应该清除任何一个你已经应用到你body上的力. 这用命令b2World::ClearForces来完成. 这让你对多个子步骤使用相同的力场.

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:

世界对象是一个包含bodies, contact, joint的容器. 你能攫取出世界的body, contact, joint清单, 并且迭代遍历他们. 例如, 这段代码唤醒世界中所有的body.

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:

不幸的是, 显示的程序可能更复杂. 例如, 下面的代码被破坏了:

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.

一切工作良好, 直到一个body被销毁. 一旦body被销毁, 它下一个指针变得无效的. 所以调用b2Body::GetNext()将返回垃圾. 解决方案是在被销毁之前复制下一个指针.

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.

这是安全的销毁当前body. 然而, 你可能想要调用一个游戏函数销毁多个body. 在这种情况下, 你要非常小心了. 解决方法是取决于应用程序, 但是为了方便, 我将展示一种解决问题的方法.

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(); // 重新获得列表指针

}

}

}

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

显然为了使这正常工作, GameCrazyBodyDestroy必须对它已经删除的body是可靠的.

### 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.

有时你需要在一个区域内确定全部的shape. b2World类有一个快速的log(N)方法适合于这种情况, 那就是使用broad-phase数据结构. 你提供在世界坐标中提供一个AABB和b2QueryCallback的实现. 对于每次查询的AABB与fixture AABB重叠, 世界对象就会调用你的类.返回true会继续查询, 否则返回false. 例如, 下面的代码寻找所有可能与一个指定AABB相交的fixture, 并且唤醒所有关联的body.

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.

关于回调的顺序你不能做任何的假定.

### 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.

你可以使用光线投影做视角检测, 打枪, 等等. 你通过实现一个回调类和提供起始与结束点, 执行一次光线投影. 对于每个被光线击中的fixture, 世界类都会调用一次你的类. 你的回调会提供fixture, 交点, 单位法向量, 沿着光线的分数距离. 你不能假设回调函数的顺序.

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.

你通过返回的分数控制射线的延长部分. 返回的分数为0表明光线投影被中断了. 分数为1表明射线投影继续犹如没有碰撞发生. 如果你返回的分数来自参数列表, 那么光线将被当前的交点裁剪. 所以你能光线投影到任何shape, 光线投影所有shape, 或者光线投影最近的shape, 通过返回适当的分数.

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.

你也可以返回-1过滤fixture. 那时光线投影处理犹如fixture不存在一样.

Here is an example:

这里是一个例子:

// 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.

注意:

应归于四舍五入的错误, 在你的静态环境里, 光线投影能够偷偷穿过多边形之间的小缝隙. 如果这在你的应用程序中是不可忍受的, 请稍微扩大你的多边形.

### 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.

你能够应用力, 扭矩, 冲量到一个body. 当你应用一个力或者一个冲量的时候, 你提供加载被应用的世界点. 这通常导致相对于重心会有个扭矩.

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.

应用力, 扭矩, 或者冲量会唤醒body. 有时这并不受欢迎. 例如, 你可能是应用一个恒定的力, 同时想给予body到sleep, 以改善执行效率. 在这种情况下你可以使用以下代码:

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).

body类有些能力帮助你转换点和向量, 在本地与世界控件之间. 如果你不明白这些概念, 请阅读” Essential Mathematics for Games and Interactive Applications” Jim Van Verth and Lars Bishop著. 这些函数是很高效的(当inlined时).

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.

你能够迭代遍历一个body的所有fixtures. 如果你需要访问fixture的userData, 这是很有用的.

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.

你能类似的迭代遍历body的joint列表.

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.

Body也提供一个关联的contact列表. 你能够用来获得关于当前contact的信息. 请小心, 接触列表可能并不包含所有contact, 因为一些contact存在于前一个时间步期间.

# Loose Ends(杂项)

## Implicit Destruction(隐式解构)

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.

Box2D不使用应用计数. 所以如果你销毁一个body, 它是实际的销毁. 访问一个指针去销毁body是没有被定义的行为. 换句话说, 你的程序将很有可能崩溃和烧毁. 为了帮助解决这些问题, 调试时建造内存管理填充销毁的实体, 使用0xFDFDFD. 这能在某些情况下更容易发现问题.

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.

如果你销毁一个Box2D实体, 那么它相当于你确保你移除所有的引用以销毁对象. 如果你使用对实体使用单一的引用, 那么这是很容易的. 如果你有多个引用, 你可能考虑实现一个句柄(handle)类以包裹未被引用的指针.

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.

通常当使用Box2D时, 你会建立和销毁很多body, shape, joint.Box2D管理这些实体是有点自动化的. 如果你销毁一个实体, 然后所有关联的shape和joint都是自动的销毁. 这叫做隐式解构.

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:

当你销毁一个body的时候, 所有它关联的shape, joint, contact都被销毁. 这叫做隐式解构. 任何body连接到那些中的一个joint和/或contact都会被唤醒. 这个处理通常是很方便的. 然而, 你必须意识到重要的问题:

**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.

注意:

当一个body被销毁时, 所有关联到此body的fixture和joint都将自动被销毁.你必须取消那些持有的shape和joint指针. 否则, 如果你在之后试图访问或者销毁这些shape或joint, 你的程序将会完全死亡.

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

为了帮助你取消你的joint指针, Box2D提供一个侦听器类, 名字是b2DestructionListener, 你能实现和装备到你的世界对象. 那么当一个joint将被隐式销毁时, 世界对象将会通知你.

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.

注意, 当一个joint或者fixture被显示的销毁时, 没有通知.在这种情况下所有权被清除, 并且你能在现场执行必要的清扫工作. 如果你喜欢, 你能调用你拥有的b2DestructioinListener实现以保证清扫工作代码集中.

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.

隐式解构在很多情况下是很便利的. 但它也能使你的程序瓦解.你可能在你代码的某处存储了shape和joint的指针. 这些指针变为孤立的, 当一个关联的body被销毁时. 情况变得更糟糕, 当你认为通过一小段代码创建的joint与关联body的管理没有联系时.例如: testbed创建一个b2MouseJoint, 用来在屏幕上交互body的操作.

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.

Box2D提供一个回调机制以通知你的应用程序, 当隐式解构发生时.这给你应用程序一个会去取消孤立的指针. 回调机制在这个手册随后的部分会有描述.

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.

你能实现一个b2DestructionListener,因为当一个body被销毁, 其关联的shape或者joint被隐式解构时, 它允许b2World通知你.

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.

你能使用你的世界对象注册一个你的b2DestructionListener实例. 你应该在世界初始化时完成这个工作.

myWorld->SetDestructionListener(myDestructionListener);

# Debug Drawing

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

你可以实现b2DebugDraw类, 以获得绘制物理世界的细节. 下面是一些有用的实体:

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



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.

这是绘制物理实体的首选方法, 而不是直接的访问数据. 原因是有很多必要的数据在内部, 同时又受制于改变.

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.

Testbed绘制物理实体使用测试绘制功能和接触侦听器, 所以它充当如何实现测试绘制和接触点绘制的主要例子.

# Limitations局限性

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

Box2D使用若干个近似法高效模拟钢体物理. 这带来了一些局限性.

Here are the current limitations:

这里是当前的局限性:

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.
7. 堆积大量的body在更轻的物体之上是不稳定的. 稳定性随着质量系统超过10:1而降低.
8. 如果一个很轻的物体支撑一个很重的物体, 通过joint连接的body链可能会被拉伸.例如: 一个破碎球连接到一个轻重量的body, 可能会变得不稳定.稳定性随着质量系统超过10:1而降低.
9. 在shape相对shape碰撞中, 有典型的大约0.5cm的误差.
10. 连续的碰撞不会处理joint. 所以你可以看见joint在快速移动对象上伸展.
11. Box2D使用偶对的欧拉积分设定. 它不复制导弹的抛物线运动, 并且只有一级精准性. 不管怎样, 它快而切有好的稳定性.
12. Box2D使用迭代求解程序提供实时执行. 你将不会获得精确的钢体碰撞或者完美的像素精度. 增加迭代次数将改善精度.

# 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