1. Box2D v2.1.0用户手册翻译 - 目录，第01章 导言(Introduction)

网上已经有个Box2D用户手册的翻译，但是基于v2.0.1，跟最新手册有很多不对应。

在这里决定将文档的全文再翻译出来，更准确的说是根据网上流传的v2.0.1版本，将最新文档重新整理一遍。

很多内容是直接复制自

Aman JIANG(江超宇)翻译的Box2D v2.0.1 用户手册

Box2D v2.1.0 用户手册

第01章 导言(Introduction)

1.1 关于

Box2D是个二维刚体仿真库, 用于编写游戏。程序员可以使用它, 让游戏中的物体运动起来更真实, 让游戏世界更具交互性。以游戏的角度来看，物理引擎只是个程序性动画系统。(procedural animation)

(译注: 做动画常有两种方法, 一种是预先准备好动画所需的数据，比如图片，再一帧一帧地播放。另一种是以一定方法，动态计算出动画所需的数据，根据数据再进行绘图。

从这种角度看，预先准备的，可称为数据性动画，动态计算的可称为程序性动画。

这个区别，就类似以前我们做历史题和数学题，做历史题，记忆很重要，也就是答案需要预先准备好的。做数学题，方法就很重要，答案是需要用方法推导出来的。

Box2D就是用物理学的方法，推导出那游戏世界物体的位置，角度等数据。而Box2D也仅仅推导出数据，至于得到数据之后怎么处理就是程序员自己的事情了。)

Box2D用可移植的C++来写成，它定义的大部分类型都有b2前缀, 希望这能有效消除Box2D和你自己的游戏引擎之间的名字冲突。

1.2 先备条件(Prerequisites)

在此手册中,我假定你已经熟悉了基本的物理概念,比如质量(mass),力(force),扭矩(torque)和冲量

(impulses)。如果没有, 建议读一下Chris Hecker和David Baraff (google 这些名字)的教程, 你不需要理解得非常细致, 只需很好地了解一些基本概念, 帮助你使用Box2D。

Wikipedia也是个很好的地方，去获取物理和数学知识。Wikipedia的内容经过了精心的整理，在某些方面可能比google更有用。

在Game Developer Conference上, Box2D是作为物理教程的一部分而创建的。你可以从box2d.org的下载区得到这些教程。

Box2D是使用C++写成的, 因此也假定你具备C++编程经验。Box2D不应该是你的第一个C++程序项目。你应该能熟练地编译,链接和调试。

注意

Box2D不应该是你的第一个C++项目。在使用Box2D之前, 请先学习C++程序设计, 还要学习怎么去编译, 连接和调试。网上有很多这方面的资料。

1.3 关于本手册

本手册涵盖了大多数Box2D的API，但并非每个方面都涉及到。Box2D自带了testbed例子, 鼓励你去看看, 以便了解更多。另外, Box2D的代码注释已被整理过, 符合Doxygen程序的格式要求, 所以很容易就可以创建一个有超链接的API文档。

1.4 反馈及错误报告

如果你想反馈Box2D的任何内容，请在论坛里留下意见。这也是个交流讨论的好去处。

Box2D使用了Google code project进行问题跟踪。这是个跟踪问题的好方法, 保证你的反馈不会被淹没在论坛深处而无人理会。

反馈地址: http://code.google.com/p/box2d/

你的问题描述得越详细,就越有可能得到修复。假如有个测试例子将问题重现,就更好了。

1.5 核心概念(Core Concepts)

Box2D中有一些基本对象,这里我们先做一个简要的定义,随后的文档会有更详细的描述。

形状(shape)
2D几何对象,比如圆形(circle)或多边形(polygon)。

刚体(rigid body)
十分坚硬的物质,坚硬得像钻石，它上面任意两点之间的距离都保持不变。在后面的讨论中,我们用物体(body)来代替刚体。

夹具(fixture)
fixture将形状绑定到物体之上,并有一定的材质属性,比如密度(density),摩擦(friction)和恢复(restitution)。

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

约束(constraint)
约束是个物理连接,用于消除物体的自由度。在2D中,物体有3个自由度(水平，垂直，旋转)。如果我们把一个物体钉在墙上(像钟摆那样),那就把它约束到了墙上。这个时候,此物体就只能绕着钉子旋转,所以这个约束消除了它2个自由度。

(译注:简单的说,需要用几个参数来确定物体的空间状态,这个物体就有几个自由度。在二维中，完全没有约束的条件下,我们要确定物体的状态,要有x坐标,y坐标,旋转角这三个参数,所以自由度为3。如果物体被钉在墙上,只要有旋转角，就可以完全确定物体的状态，有了钉子这个约束，物体自由度就变成了1。)

接触约束(contact constraint)
一种特殊的约束,设计的目的是为了防止刚体被穿透,也用于模拟摩擦和恢复。接触约束不用你来创建,它们会自动被Box2D生成。

关节(joint)
关节就是种约束,用于将两个或多个body固定到一起。Box2D支持不同的关节类型:转动(revolute),棱柱(prismatic),距离(distance)等。一些关节可以有限制(limits)和马达(motors)。

关节限制(joint limit)
关节限制限定了一个关节的运动范围。例如人类的胳膊肘只能在某一角度范围内运动。

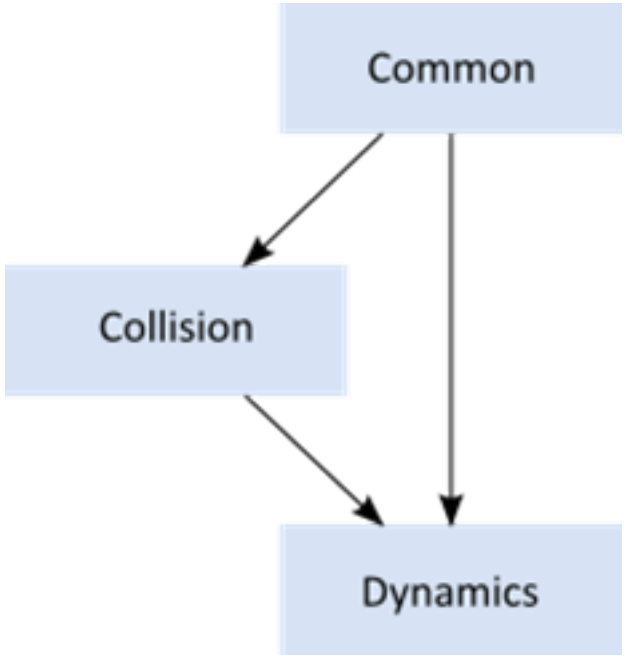
关节马达(joint motor)
根据关节的自由度, 关节马达可以驱动关节所连接的物体。例如， 你可以使用一个马达来驱动一个肘的旋转。

世界(world)
一个物理世界就是各种, 刚体(bodies), 夹具(fixtures), 约束(constraints)相互作用的集合。 Box2D支持创建多个世界, 但这通常没有必要。

1.6 模块(Modules)

Box2D由三个模块组成：公共(Common)、碰撞(Collision)、和动态(Dynamics)。Common模块包括了内存分配、数学计算和配置。Collision模块定义了形状(shapes)、broad-phase检测和碰撞功能/查询(collision functions/queries)。最后， Dynamics模块提供对世界(world)、刚体(bodies)、夹具(fixtures)和关节(joint)的模拟。

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



1.7 单位

Box2D使用浮点数, 所以必须使用一些公差来保证它正常工作。这些公差已经被调谐得适合米-千克-

秒(MKS)单位。尤其是, Box2D被调谐得能良好地处理0.1到10米之间的移动物体。这意味着从罐头盒到公共汽车大小的对象都能良好地工作。静态的物体就算到50米都没有大问题。

作为一个2D物理引擎，如果能使用像素作为单位是很诱人的。很不幸，那将导致不良模拟，也可能会造成古怪的行为。一个200像素长的物体在Box2D看来就有45层建筑那么大。想象下使用一个已调谐好用于模拟玩偶和木桶的引擎去模拟高楼大厦的运动。那并不有趣。

注意

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

最好把Box2D中的物体看作移动的广告板, 其上带着你的艺术创作。广告板在一个以米为单位的系统里运动，但你可以利用简单的比例因子把它转换为像素坐标。之后就可以使用这些像素坐标去确定你的精灵(sprites)的位置，等等。

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

1.8 工厂和定义

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

这些是创建函数:

b2Body* b2World::CreateBody(constb2BodyDef* def)

b2Joint* b2World::CreateJoint(constb2JointDef* def)

这些是对应的摧毁函数:

void b2World::DestroyBody(b2Body*body)

void b2World::DestroyJoint(b2Joint*joint)

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

fixture必须有父body, 要使用b2Body的工厂方法来创建及摧毁。

b2Fixture* b2Body::CreateFixture(constb2FixtureDef* def)

voidb2Body::DestroyFixture(b2Fixture* fixture)

也有个简便方法直接用形状和密度来创建fixture

b2Fixture* b2Body::CreateFixture(constb2Shape* shape, float32 density)

工厂并不保留定义的引用, 你可以在栈上临时创建定义。

1.9 用户数据

b2Fixture, b2Body 和 b2Joint 类都允许你通过一个 void 指针来附加用户数据。当你测试Box2D, 以及使得Box2D的数据结构跟自己的游戏引擎结合起来, 用void指针是较为方便的。

举个典型的例子, 角色上有刚体, 并在刚体中附加角色的指针, 就构成了一个循环引用。如果你有角色(actor), 你就能得到刚体。如果你有刚体,你就能得到角色。

```
GameActor* actor =GameCreateActor();

b2BodyDef bodyDef;

bodyDef.userData = actor;

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

一些需要用户数据的例子:

- 使用碰撞结果给角色施加伤害。
- 当玩家进入一个包围盒(axis-aligned box)时触发脚本事件
- 当Box2D通知关节就要摧毁时, 去访问某个游戏结构。

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

2. Box2D v2.1.0用户手册翻译 - 第02章 Hello Box2D

内容很多摘自

Aman JIANG(江超宇)翻译的Box2D v2.0.1 用户手册

第02章 Hello Box2D

Box2D的发布包中有个Hello World程序。程序创建了一个大大的地面盒(ground box)和一个小小的动态盒(dynamic box)。盒子的位置随着时间的变化而变化。代码没有涉及到图形界面，你只能在控制台中看到文字输出

这是个很好的例子, 展示了怎么使用Box2D。

2.1 创建世界(Creating a World)

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

创建Box2D的world很简单。首先, 我们要定义重力矢量，另外还要告诉world是否允许body在静止时休眠。休眠中的body不需要任何模拟。

```
b2Vec2 gravity(0.0f, -10.0f);
```

```
bool doSleep = true;
```

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

```
b2World world(gravity, doSleep);
```

我们已经有了自己的物理世界, 开始向里面加东西了。

2.2 创建地面盒(Creating a Ground Box)

body用以下步骤来创建：

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

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

```
b2BodyDef groundBodyDef;

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

第二步, 将body定义传給world对象, 创建ground body。world对象并不保留body定义的引用。ground body是作为静态物体(static body)创建的。静态物体和其它静态物体之间并没有碰撞, 它们是固定的。当body的质量为零时, Box2D就认为它是静态的。物体质量的默认值就为零, 所以它们默认就是静态的。

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

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

b2PolygonShape groundBox;

groundBox.SetAsBox(50.0f, 10.0f);

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

第四步, 我们创建shape fixture, 以完成ground body。这步中，我们有个简便方法。我们并不需要修改fixture默认的材质属性, 可以直接将形状传给body而不需要创建fixture的定义。随后的教程中, 我们将会看到如何使用fixture定义来定制材质属性。

groundBody->CreateFixture(&groundBox);

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

注意，每个fixture都必须有一个父body，即使fixture是静态的。然而，你可以把所有静态fixture都依附于单个静态body之上。之所以需要这个静态body, 是为了保证Box2D内部的代码更具一致性，以减少潜在的bug数量。

可能你已经注意到, 多数Box2D类型都有b2前缀。这是为了降低它和你的代码之间名字冲突的机会。

2.3 创建动态物体(Creating a Dynamic Body)

现在我们已经有了一个地面body,我们可以使用同样的方法来创建一个动态body。除尺寸之外的主要

区别是, 我们必须为动态body设置质量属性。

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

```
b2BodyDef bodyDef;

bodyDef.type = b2_dynamicBody;

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

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

注意

如果你想body受力的影响而运动, 你必须将body的类型设为b2_dynamicBody。

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

```
b2PolygonShape dynamicBox;

dynamicBox.SetAsBox(1.0f, 1.0f);
```

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

b2FixtureDef fixtureDef;

fixtureDef.shape = &dynamicBox;

fixtureDef.density = 1.0f;

fixtureDef.friction = 0.3f;

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

body->CreateFixture(&fixtureDef);

这就是初始化过程。现在我们已经做好准备，可以开始模拟了。

2.4 模拟(Box2D的)世界

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

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

float32 timeStep = 1.0f / 60.0f;

除积分器外,Box2D代码还使用了约束求解器(constraint solver)。约束求解器用于解决模拟中的所有约

束,一次一个。单个的约束会被完美的求解,然而当我们求解一个约束的时候,我们就会稍微耽误另一个。要得到良好的解,我们需要多次迭代所有约束。

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

对于速度和位置，建议的Box2D迭代次数都是10次。你可以按自己的喜好去调整这个数字，但要记得它是性能与精度之间的折中。更少的迭代会增加性能但降低精度，同样地，更多的迭代会降低性能但能提高模拟质量。对于这个简单示例，我们不需要多次迭代。这是我们选择的迭代次数。

```
int32 velocityIterations = 6;

int32 positionIterations = 2;
```

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

现在我们可以开始模拟循环了, 在你的游戏中, 模拟循环和游戏循环可以合并起来。每次游戏循环你都应该调用b2World::Step, 通常调用一次就够了, 这取决于帧频以及物理时间步。步进后，你应当调用b2World::ClearForces清除你施加到body上的任何力。

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

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

{

        world.Step(timeStep,velocityIterations, positionIterations);

        world.ClearForces();
```

```
        b2Vec2 position =body->GetPosition();

        float32 angle =body->GetAngle();

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

}
```

输出展示了动态box降落到地面的情况。你的输出看起来应当是这样：

```
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
```

2.5 清理

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

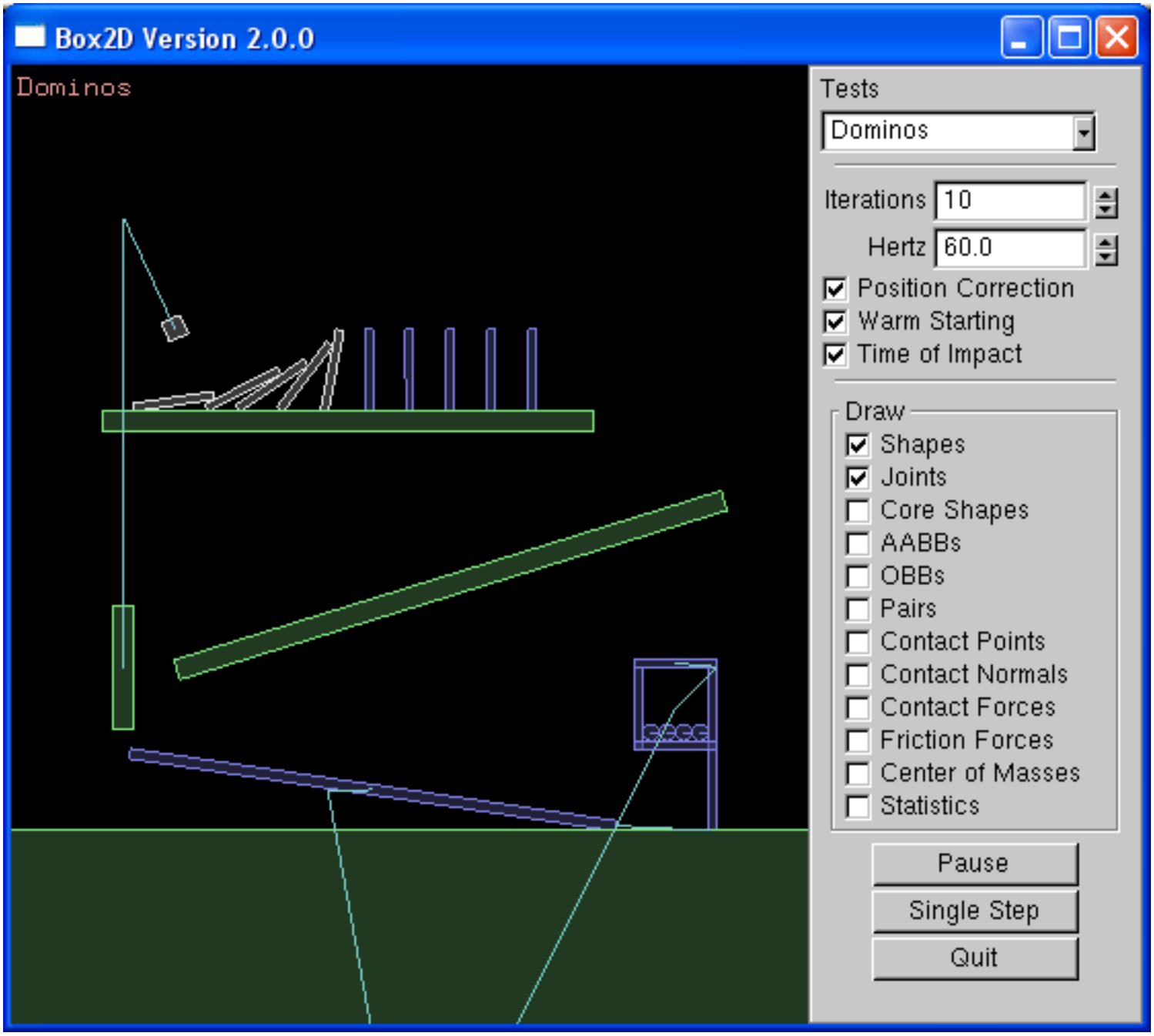
2.6 Testbed例子

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

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

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

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



3. Box2D v2.1.0用户手册翻译 - 第03章 公共模块(Common)

内容很多摘自

Aman JIANG(江超宇)翻译的Box2D v2.0.1 用户手册

第03章 公共模块(Common)

3.1 关于

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

3.2 配置

头文件 b2Settings.h 包含:

- 类型, 比如int32和float32
- 常数
- 分配器包装(Allocation wrappers)
- 版本号
- 摩擦混合及恢复混合的函数

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

常数
Box2D定义了一些常数, 都记录在b2Settings.h中。通常你不需要调整这些常数。

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

分配器包装
配置文件定义了b2Alloc和b2Free, 用于大内存的分配。你可以让b2Alloc和b2Free再调用自己的内存管理系统(memory management system)。

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

摩擦混合及恢复混合
假如你的应用程序需要定制这些混合函数, 可以在配置文件中找到它们。

3.3 内存管理

Box2D 的许多设计都是为了能快速有效地使用内存。在本节我将论述 Box2D 如何及为什么要这样分配内存。

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

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

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

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

3.4 数学

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

数学库保持得尽量简单, 使得Box2D容易移植和维护。

4. Box2D v2.1.0用户手册翻译 - 第04章 碰撞模块(Collision Module)

第04章 碰撞模块(Collision Module)

4.1 关于

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

4.2 形状(Shapes)

形状描述了可相互碰撞的几何对象, 就算不进行物理模拟，也可独立使用。你可以在shape上执行一些操作。

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

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

另外, 每个形状都有两成员变量: 类型(type)和半径(radius)。 对于多边形，半径也是有意义的, 下面会进行讨论。

4.3 圆形(Circle Shapes)

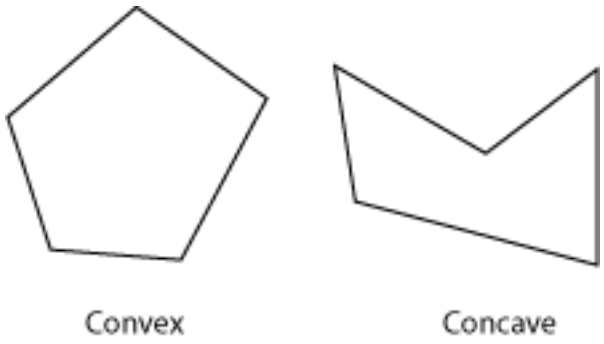
圆形有位置和半径。

圆形是实心的，你没有办法使圆形变成空心。但是，你可以使用多边形来创建一系列线段，让这些线段首尾相连，串成一串，就可以模拟出空心的圆形。

```
b2CircleShape circle;

circle.m_p.Set(1.0f, 2.0f, 3.0f);

circle.m_radius = 0.5f;
```

4.4 多边形(Polygon Shapes)

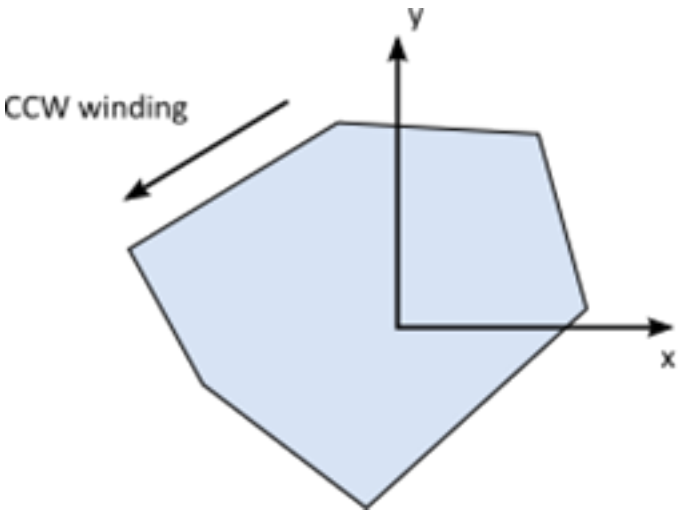
Box2D的多边形是实心的凸(Convex)多边形。在多边形内部任意选择两点，作一线段，如果所有的线段跟多边形的边都不相交，这个多边形就是凸多边形。多边形是实心的，不可能空心。但是，你可以使用两个点来创建多边形，这样就退化成线段。



创建多边形时，使用的点必须是逆时针排列(CCW)。我们必须很小心，逆时针是相对于右手坐标系统来说的，这坐标系下，Z轴指向平面外面。有可能相对于你的屏幕，就变成顺时针了，这取决于你自己的坐标系统是怎么规定的。

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

创建多边形时，你可以传递一个包含顶点的数组。数组大小最多是b2_maxPolygonVertices，这数值默认是8。这已足够描述大多数的凸多边形了。



// 按逆时针顺序定义一个三角形

b2Vec2 vertices[3];

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

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

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

int32 count = 3;

b2PolygonShape polygon;

polygon.Set(vertices, count);

多边形有一些定义好的初始化函数来创建箱(box)和边(edge,也就是线段)。

void SetAsBox(float32 hx, float32hy);

void SetAsBox(float32 hx, float32hy, const b2Vec2& center, float32 angle);

void SetAsEdge(const b2Vec2& v1,const b2Vec2& v2);

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

(译注：这一段我不太明白，原文是

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.)

4.5 **形状的点测试**(Shape Point Test)

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

b2Transfrom transform;

transform.SetIdentity();

b2Vec2 point(5.0f, 2.0f);

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

(译注: Box2D中，形状附加在物体之上，它存储的数据是在物体的局部坐标系下定义的，而传进来要

测试的点是在世界坐标系下，坐标系不同，就没有办法比较。这个transform用于将形状从局部坐标系转到世界坐标系，之后才可进行比较。而transform的逆转换就是将世界坐标系转到局部坐标系。故要实现这个函数，也可以先求逆转换，将传过来的点转到局部坐标系，这同样可以进行比较，要看哪一个方便。

看Box2D的源码，它实现b2CircleShape::TestPoint时，是将圆心转成世界坐标系，再比较。而b2PolygonShape::TestPoint，是将传进来的点先转成局部坐标再比较。因为转成世界坐标，多边形要同时转换多个点，而圆形就只转换圆心。

使用局部坐标系，不管物体怎么移动，旋转及缩放，改变的只是这个转换矩阵，形状存储的点不用修改，这样就很方便了。下面文档中，你可以看到形状的很多函数，都会传进一个转换，道理是一样的。)

4.6 形状的光线投射(Shape Ray Cast)

你可以用光线射向形状，得到它们之间的交点和法向量。如果在形状内部开始投射，就当成没有交点，返回false。

```
b2Transfrom transform;

transform.SetIdentity();

b2RayCastInput input;

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

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

input.maxFraction = 1.0f;

b2RayCastOutput output;
```

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

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。）

4.7 对等函数(Bilateral Functions)

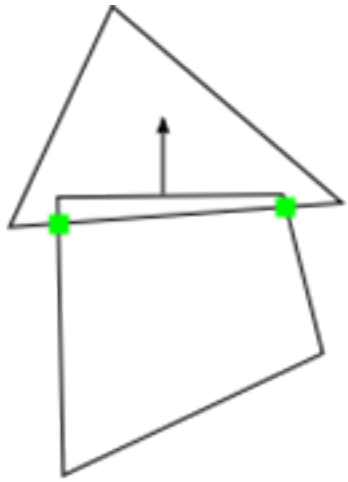
碰撞模块含有对等函数，要传递两个形状，计算出结果。包括:

- 接触manifolds
- 距离
- 撞击时间

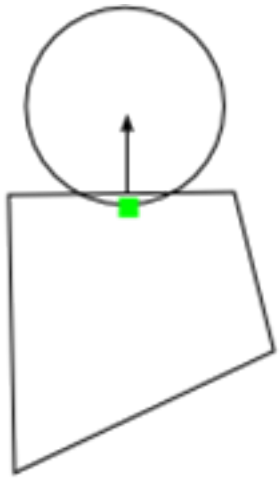
4.8 接触Manifolds

(译注: 我不知道Manifold应该翻译成什么，我猜Manifold指有相同属性的一堆东西，可以理解成集合，但翻译成集合又不好。故直接保留英文)

Box2D有些函数用来计算重合形状之间的接触点。考虑一下圆与圆，圆与多边形的碰撞，我们只会得到一个接触点和一个向量。多边形与多边形的碰撞，我们可以得到两个接触点。这些接触点都具有相同的法向量，所以Box2D就将它们归成一组，构成manifold结构。接触求解器进一步处理这结构，以改善物体堆叠在一起时，系统的稳定性。



two points, one normal



two points, one normal

通常你不需要直接计算接触manifold, 但你可能会希望使用这模拟过程中已处理好的结果。

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

b2WorldManifold结构可以用来生成世界坐标下的接触向量和点。你需要提供b2Manifold结构和形状的转换及半径。

```
b2WorldManifold worldManifold;

worldManifold.Initialize(&manifold,transformA, shapeA.m_radius,
```

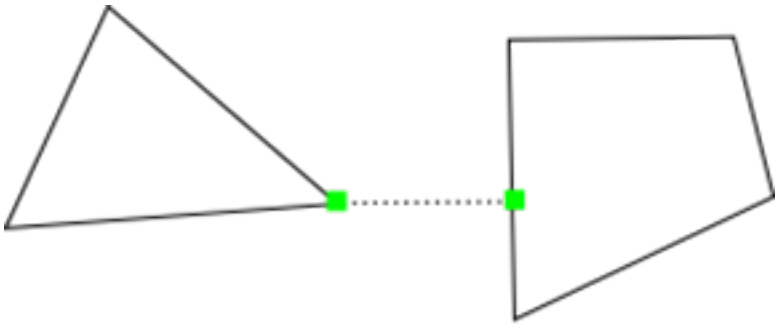
```
                                        transformB,shapeB.m_radius);

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

{

        b2Vec2 point =worldManifold.points[i];



}
```

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

```
b2PointState state1[2], state2[2];

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

if (state1[0] == b2_removeState)

{

        // process event

}
```

4.9 距离(Distance)

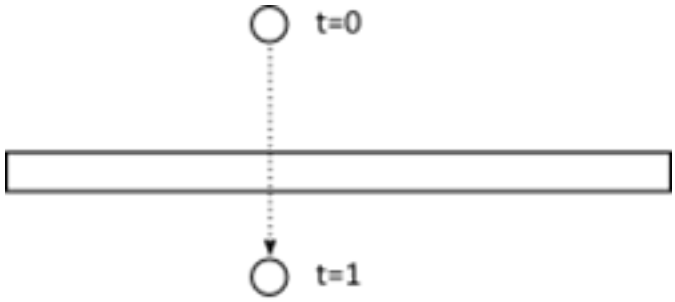
b2Distance函数可以用来计算两个形状之间的距离。距离函数需要两个形状，转成b2DistanceProxy。There is also some caching used to warm start the distance function for repeated calls.(看不明白，见谅)。详细见b2Distance.h文件。



distance and closest points

4.10 撞击时间(Time of Impact)

如果两个形状快速移动，它们可能会在一个时间步内穿过对方。



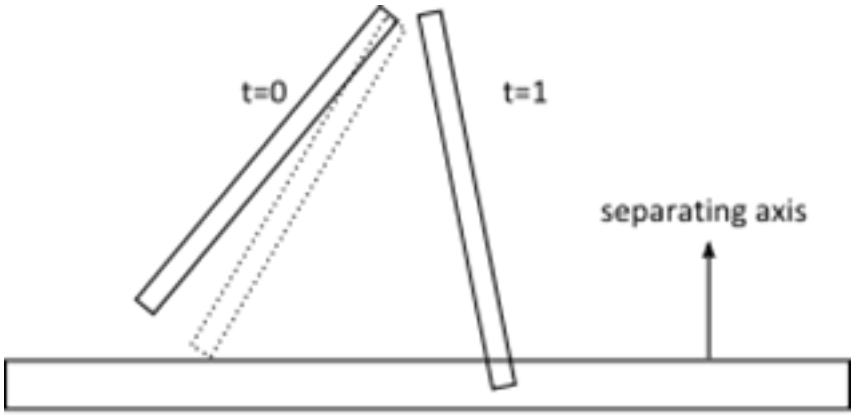
tunneling

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

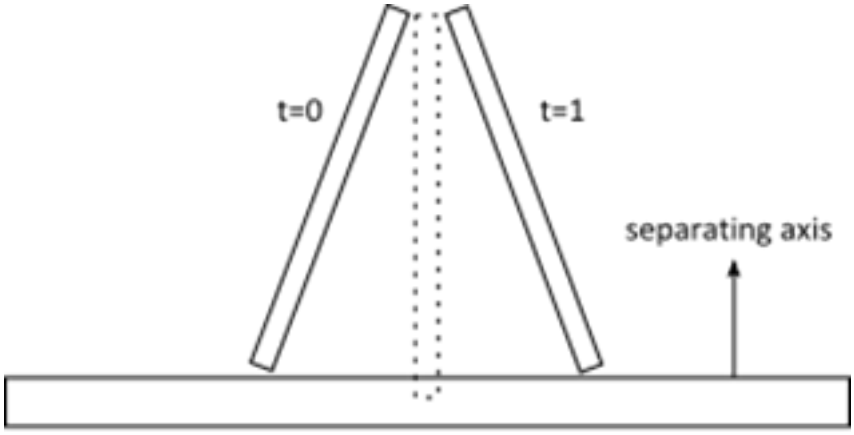
这个函数考虑了形状的旋转和平移，但如果旋转足够大，这函数还是会错过碰撞。函数会报告一个非重叠的时间，并捕捉到所有的平移碰撞。

撞击时间函数最开始时定义了一条的分离轴，并确保形状没有超过这条轴。这可能会在结束位置错

过一些碰撞。但这方法很快，在防止隧穿方面也已经足够了。



captured collision



missed collision

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

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

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

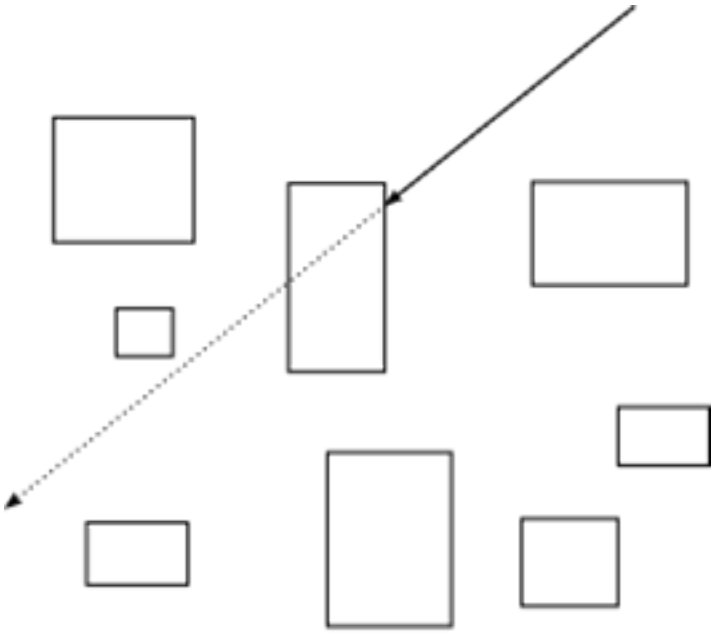
4.11 动态树(Dynamic Tree)

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

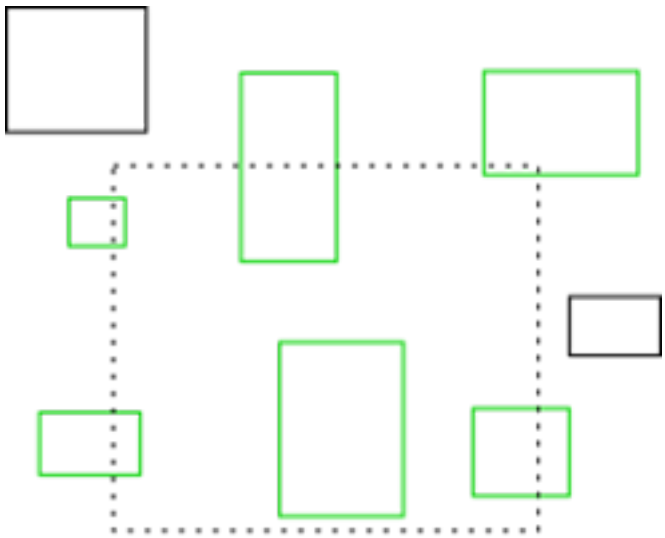
动态树是分层的AABB树。树的每个内部节点都有两个子节点。左边的子节点是用户的AABB。

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

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



ray cast



region query

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

4.12 Broad-phase

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

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

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

5. Box2D v2.1.0用户手册翻译 - 第05章 动态模块(Dynamics Module)

第05章 动态模块(Dynamics Module)

5.1 概述

动态模块是Box2D中最复杂的部分，你与这模块之间的交互也最多。动态模块构建在通用和碰撞模块的基础上，到现在你对这两个模块也应该有所了解了。

动态模块包括下面这些类:

- 形状(shape), 夹具(fixture)
- 刚体
- 接触
- 关节
- 世界
- 监听器(listener)

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

动态模块包括接下来的章节。

6. Box2D v2.1.0用户手册翻译 - 第06章 夹具(Fixtures)

内容很多摘自

Aman JIANG(江超宇)翻译的Box2D v2.0.1 用户手册

第06章 夹具(Fixtures)

6.1 关于

回想一下，形状并不知道物体的存在，可以独立使用。因此Box2D需要提供b2Fixture类，用于将形状附加到物体上。 fixture具有下列属性:

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

这些都会在接下来的小节中描述。

6.2 创建夹具(Fixture Creation)

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

```
b2FixtureDef fixtureDef;

fixtureDef.shape = &myShape;

fixtureDef.density = 1.0f;

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

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

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

```
myBody->DestroyFixture(myFixture);
```

密度(Density)
fixture的密度用来计算父物体的质量属性。密度值可以为零或者是整数。你所有的fixture都应该使用相似的密度，这样做可以改善物体的稳定性。

当你添加一个fixture时，物体的质量会自动调整。

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

float32 friction;

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

所以当其中一个fixture的摩擦参数为0时，接触的摩擦就为0。

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

float32 restitution;

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

ixture会携带碰撞筛选信息，让你防止某些游戏对象相互碰撞。

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

筛选(Filtering)
碰撞筛选是一个防止某些形状发生碰撞的系统。比如,你创造了一个骑自行车的角色。你希望自行车与地形之间有碰撞,角色与地形有碰撞,但你不希望角色和自行车之间发生碰撞 (因为它们必须重叠)。Box2D通过种群和分组支持了这样的碰撞筛选。

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

playerFixtureDef.filter.categoryBits= 0x0002;

monsterFixtureDef.filter.categoryBits= 0x0004;

playerFixtureDef.filter.maskBits =0x0004;

monsterFixtureDef.filter.maskBits =0x0002;

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

fixture1Def.filter.groupIndex = 2;

fixture2Def.filter.groupIndex = 2;

fixture3Def.filter.groupIndex = -8;

fixture4Def.filter.groupIndex = -8;

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

注意在 Box2D 中还有其它的碰撞筛选,这里是一个列表:

- static物体上的fixture永远不会与另一个static体上的fixture发生碰撞
- 同一个物体上的fixture永远不会相互碰撞
- 如果两个物体用关节连接起来，物体上面的fixture可以选择启用或禁止它们之间相互碰撞

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

6.3 传感器(Sensors)

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

你可以将任一fixture标记为传感器。传感器可以是static或dynamic的。记住,每个物体上可以有多个fixture, 传感器和实体fixture是可以混合存在的。

传感器不会生成接触点。这里有两种方法得到传感器的状态:

1. b2Contact::IsTouching

2. b2ContactListener::BeginContact 和 EndContact

7. Box2D v2.1.0用户手册翻译 - 第07章 物体(Bodies)

内容很多摘自

Aman JIANG(江超宇)翻译的Box2D v2.0.1 用户手册

第07章 物体(Bodies)

7.1 关于

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

b2_staticBody
static物体在模拟时不会运动，就好像它具有无穷大的质量。在Box2D内部，会将static物体的质量存储为零。static物体可以让用户手动移动，它速度为零，另外也不会和其它static或kinematic物体相互碰撞。

b2_kinematicBody
kinematic物体在模拟时以一定速度运动, 但不受力的作用。它们可以让用户手动移动，但通常的做法是设置一定速度。kinematic物体的行为表现就好像它具有无穷大的质量，Box2D将它的质量存储为零。kinematic物体并不会和其它static或kinematic物体相互碰撞。

b2_dynamicBody
dynamic物体被完全模拟。它们可以让用户手动移动，但通常它们都是受力的作用而运动。
dynamic物体可以和其它所有类型的物体相互碰撞。dynamic物体的质量总是有限大的，非零的。如果你试图将它的质量设置为零，它会自动地将质量修改成一千克。

物体是fixtures的骨架，带着fixture在世界中运动。Box2D中的物体总是刚体(rigid body)。也就是说，同一刚体上的两个fixture，永远不会相对移动。

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

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

7.2 物体定义(Body Definition)

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

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

让我们看一些物体定义的关键成员。

物体类型(Body Type)

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

```
bodyDef.type = b2_dynamicBody;
```

物体类型是一定要设置的。

位置和角度(Position and Angle)

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

注意

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

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

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

```
bodyDef.position.Set(0.0f, 2.0f);// body的原点
```

```
bodyDef.angle = 0.25f * b2_pi;// 弧度制下body的角
```

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

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

```
bodyDef.linearDamping = 0.0f;
```

```
bodyDef.angularDamping = 0.01f;
```

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

休眠参数(Sleep Parameters)
休眠是什么意思? 模拟物体的成本是高昂的,所以如果物体更少,那模拟的效果就能更好。当物体停止了运动时, 我们会希望停止模拟它。

当Box2D确定一个物体(或一组物体)已停止移动时,物体就会进入休眠状态。休眠物体只消耗很小的

CPU开销。如果一个醒着的物体接触到了一个休眠中的物体,那么休眠中的物体就会醒过来。当物体上的关节或触点被摧毁的时候,它们同样会醒过来。你也可以手动地唤醒物体。

通过物体定义,你可以指定一个物体是否可以休眠,或者创建一个休眠的物体。

```
bodyDef.allowSleep = true;
```

```
bodyDef.awake = true;
```

固定旋转(Fixed Rotation)
你可能想一个刚体，比如某个角色，具有固定的旋转角。这样物体即使在负载下，也不会旋转。 你可以设置fixedRotation来达到这目的:

```
bodyDef.fixedRotation = true;
```

固定旋转标记使得转动惯量被设置成零。

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

默认情况下,Box2D会通过连续碰撞检测(CCD)来防止动态物体穿越静态物体。这是通过扫描形状从旧 位置到新位置的过程来完成的。引擎会查找扫描中的新碰撞,并为这些碰撞计算碰撞时间 (TOI)。物体会先被移动到它们的第一个TOI,然后一直模拟到原时间步的结束。

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

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

bodyDef.bullet = true;

子弹标记只影响dynamic物体。

Box2D执行连续碰撞检测，所以子弹也有可能会错过快速移动的物体。

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

你可以创建一个非活动的物体，之后再激活它。

bodyDef.active = true;

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

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

b2BodyDef bodyDef;

bodyDef.userData = &myActor;

7.3 物体工厂(Body Factory)

物体使用world类提供的工厂来创建和摧毁。这让world可以通过一个高效的分配器来创建物体,并且把物体加入到world的数据结构中。

物体可以是dynamic或static的,这取决于质量属性。两种类型物体的创建和摧毁方法都是一样的。

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

... do stuff ...

myWorld->DestroyBody(dynamicBody);

dynamicBody = NULL;
```

注意

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

static物体不会受其它物体的作用而移动。你可以手动地移动static物体,但你必须小心,不要挤压到static物体之间的dynamic物体。另外,当你移动static物体的时候,摩擦不会正确工作。static物体不会和其它static或kinematic物体碰撞。在单个static物体上附加数个形状, 要比多个static物体都只附加单个形状，有更好的性能。在内部, Box2D会设置static物体的质量为零。这使得大部分算法都不必把静态物体当成特殊情况来看待。

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

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

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

7.4 使用物体(Using a Body)

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

质量数据(Mass Data)

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

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

```
void SetMassData(const b2MassData* data);
```

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

```
void ResetMassData();
```

要得到物体的质量数据，可以通过下面的函数:

```
float32 GetMass() const;

float32 GetInertia() const;

const b2Vec2& GetLocalCenter()const;

void GetMassData(b2MassData* data)const;
```

状态信息(State Information)
物体的有多个方面状态。你可以通过下面的函数高效地访问状态数据:

```
void SetType(b2BodyType type);

b2BodyType GetType();

void SetBullet(bool flag);

bool IsBullet() const;

void SetSleepingAllowed(bool flag);

bool IsSleepingAllowed() const;

void SetAwake(bool flag);

bool IsAwake() const;
```

```
void SetActive(bool flag);

bool IsActive() const;



void SetFixedRotation(bool flag);

bool IsFixedRotation() const;
```

位置和速度(Position and Velocity)
你可以访问一个物体的位置和旋转角, 这在你渲染相关游戏角色时很常用。通常情况下，你都是使用Box2D来模拟运动，也可以设置位置, 但不怎么常用。

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

const b2Transform&GetTransform() const;

const b2Vec2& GetPosition()const;

float32 GetAngle() const;
```

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

```
const b2Vec2& GetWorldCenter()const;
```

const b2Vec2& GetLocalCenter()const;

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

8. Box2D v2.1.0用户手册翻译 - 第08章 关节(Joints)

内容很多摘自

Aman JIANG(江超宇)翻译的Box2D v2.0.1 用户手册

第08章 关节(Joints)

8.1 关于

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

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

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

8.2 关节定义

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

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

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

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

其余的关节定义数据依赖于关节的类型。下面我们来介绍它们。

8.3 关节工厂(Joint Factory)

关节使用world的工厂方法来创建和摧毁。这引入一个老问题：

注意:

不要使用new或malloc在栈(stack)或堆(heap)中创建关节。你想创建或摧毁物体和关节，必须使用b2World类中对应的创建或摧毁函数。

这里有个例子，展示了旋转关节(revolute joint)从创建到摧毁的过程:

```
b2RevoluteJointDef jointDef;

jointDef.body1 = myBody1;

jointDef.body2 = myBody2;

jointDef.anchorPoint =myBody1->GetCenterPosition();
```

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

... do stuff ...

myWorld->DestroyJoint(joint);

joint = NULL;
```

一个很好的习惯: 当对象摧毁后，就将对应的指针清零。不清零的话，当你试图再次使用这个指针，程序在特定时候会崩溃。

关节的生命周期并不简单，要特别留心下面的警告，敲响警钟。

注意

物体被摧毁时，依附其上的关节也会被摧毁。

上面的注意并非时时必要。你可以组织好自己的游戏引擎，保证物体被摧毁前，依附其上的关节已经先被摧毁。这种情况下，你没有必要实现监听类(listener class), 去监听物体被摧毁的事件。更多细节请看隐式摧毁(Implicit Destruction)那小节。

8.4 使用关节

在许多模拟中,关节被创建之后，直到摧毁也不会再被访问。然而,关节中包含着很多有用的数据, 使你可以创建出丰富的模拟。

首先,你可以在关节上得到物体,锚点,以及用户数据。

b2Body* GetBody1();

b2Body* GetBody2();

b2Vec2 GetAnchor1();

b2Vec2 GetAnchor2();

void* GetUserData();

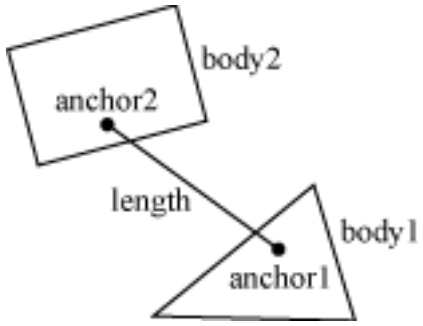
所有的关节都有反作用力和反扭矩,这个反作用力应用于 body2 的锚点之上。你可以用反作用力来折断关节(break joints),或引发其它游戏事件。这些函数可能需要做一些计算,所以没有必要就不要去调用它们。

b2Vec2 GetReactionForce();

float32 GetReactionTorque();

8.5 距离关节(Distance Joint)

距离关节是最简单的关节之一, 它是说, 两个物体上面各自有一点, 两点之间的距离必须固定不变。当你指定一个距离关节时, 两个物体必须已在应有的位置上。之后,你指定世界坐标中的两个锚点。第一个锚点连接 到物体1,第二个锚点连接到物体2。这两点隐含了距离约束的长度。



这是一个距离关节定义的例子。这种情况下, 我们允许物体碰撞。

b2DistanceJointDef jointDef;

jointDef.Initialize(myBody1,myBody2, worldAnchorOnBody1, worldAnchorOnBody2);

jointDef.collideConnected = true;

距离关节也可以是软的，就像用橡皮筋来连接。看看testbed中的Web例子，可以知道出它有什么样的行为。

要使关节有弹性，可以调节一下两个参数:频率(frequency)和阻尼率(damping ratio)。将频率想象成谐振子(harmonic oscillator, 比如吉他弦)振动的快慢。频率使用单位赫兹(Hertz)来指定。典型情况下，关节频率要小于一半的时间步(time step)频率。比如每秒执行60次时间步, 距离关节的频率就要小于30赫兹。这样做的理由可以参考Nyquist频率理论。

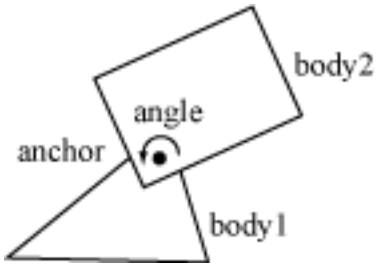
阻尼率无单位，典型是在0到1之间, 也可以更大。1是阻尼率的临界值, 当阻尼率为1时，没有振动。

jointDef.frequencyHz = 4.0f;

jointDef.dampingRatio = 0.5f;

8.6 旋转关节(Revolute Joint)

旋转关节会强制两个物体共享一个锚点,即所谓铰接点。旋转关节只有一个自由度:两个物体的相对旋转。这称之为关节角。



要指定一个旋转关节,你需要提供两个物体以及世界坐标的一个锚点。初始化函数会假定物体已经在应有位置了。

在此例中,两个物体被旋转关节连接起来, 铰接点为第一个物体的质心。

```
b2RevoluteJointDef jointDef;

jointDef.Initialize(myBody1,myBody2, myBody1->GetWorldCenter());
```

在 body2 逆时针旋转时,关节角为正。像所有 Box2D 中的角度一样,旋转角也是弧度制的。按规 定,使用Initialize() 创建关节时, 无论两个物体当前的角度怎样, 旋转关节角都为0。

有时候,你可能需要控制关节角。为此,旋转关节可以随意地模拟关节限制和马达。

关节限制(joint limit)会强制关节角度保持在一定范围内。为此它会应用足够的扭矩。0应该在范围内,否则在开始模拟时关节会有点倾斜。

关节马达允许你指定关节的角速度(角度的时间导数),速度可正可负。马达可以有产生无限大的力,但这通常是没有必要。想想那个经典问题:

"当一个不可抵抗的力作用在一个不可移动的物体上, 会发生什么?"

我可以告诉你这并不有趣。所以你应该为关节马达提供一个最大扭矩。关节马达会维持在指定的速度,除非其所需的扭矩超出了最大扭矩。当超出最大扭矩时,关节会慢下来,甚至会反向运动。

你还可以使用关节马达来模拟关节摩擦。只要把关节速度设为0,并将最大扭矩设得很小且有效。这样马达会试图阻止关节旋转, 除非有过大的负载。

这里是对上面旋转关节定义的修订; 这次,关节拥有一个限制以及一个马达,后者用于模拟摩擦。

```
b2RevoluteJointDef jointDef;

jointDef.Initialize(body1, body2,myBody1->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;
```

你可以访问旋转关节的关节角，速度，马达扭矩。

```
float32 GetJointAngle() const;

float32 GetJointSpeed() const;

float32 GetMotorTorque() const;
```

执行step后，你也可以更新马达的参数。

```
void SetMotorSpeed(float32 speed);

void SetMaxMotorTorque(float32torque);
```

关节马达有些有趣的功能。你可以在每个时间步中更新关节的速度，使得它像正弦波那样前后摆动，或者指定一个你想要的函数。

```
... Game Loop Begin ...

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

... Game Loop End ...
```

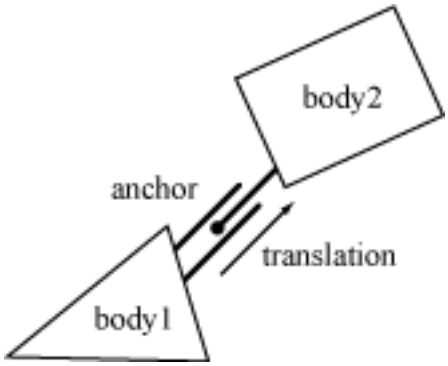
你也可以用关节马达来跟踪你想要的关节角。比如:

```
... Game Loop Begin ...

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

float32 gain = 0.1f;

myJoint->SetMotorSpeed(-gain *angleError);

... Game Loop End ...
```

通常你的增益参数不能太大，不然关节会变得不稳定。

8.7 移动关节(Prismatic Joint)

移动关节(prismatic joint)允许两个物体沿指定轴相对移动,它会阻止相对旋转。因此,移动关节只有一个自由度。



移动关节的定义有些类似于旋转关节;只是转动角度换成了平移,扭矩换成了力。以这样的类比,我们来看一个带有关节限制以及马达摩擦的移动关节定义:

```
b2PrismaticJointDef jointDef;

b2Vec2 worldAxis(1.0f, 0.0f);

jointDef.Initialize(myBody1,myBody2, myBody1->GetWorldCenter(), worldAxis);

jointDef.lowerTranslation = -5.0f;

jointDef.upperTranslation = 2.5f;

jointDef.enableLimit = true;

jointDef.motorForce = 1.0f;
```

```
jointDef.motorSpeed = 0.0f;

jointDef.enableMotor = true;
```

旋转关节隐含着一个从屏幕射出的轴,而移动关节明确地需要一个平行于屏幕的轴。这个轴会固定于两个物体之上,沿着它们的运动方向。

就像旋转关节一样,当使用 Initialize() 创建移动关节时,移动为0。所以要确保0在你的移动限制范围内。

移动关节的用法跟旋转关节类似。这是相应的函数:

```
float32 GetJointTranslation() const;

float32 GetJointSpeed() const;

float32 GetMotorForce() const;

void SetMotorSpeed(float32 speed);

void SetMotorForce(float32 force);
```

8.8 滑轮关节(Pulley Joint)

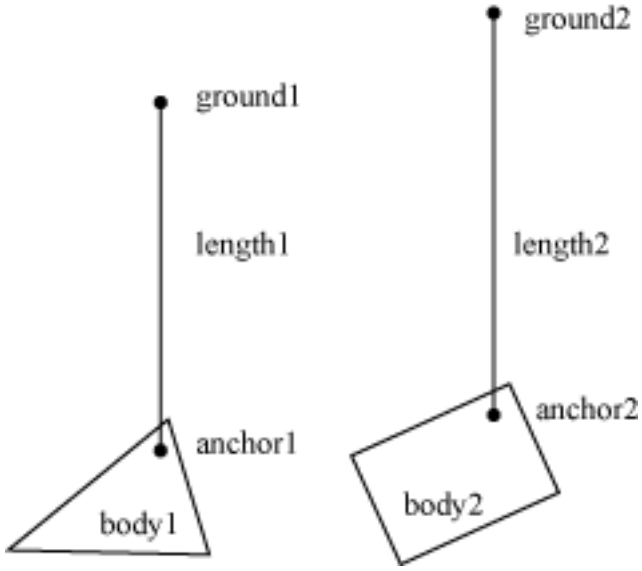
滑轮关节用于创建理想的滑轮,它将两个物体接地(ground)并彼此连接。这样,当一个物体上升,另一个物体就会下降。滑轮的绳子长度取决于初始配置。

length1 + length2 == constant

你还可以提供一个系数(ratio)来模拟block and tackle,这会使滑轮一侧的运动比另一侧要快。同时,一侧的约 束力也比另一侧要小。你也可以用这个来模拟机械杠杆(mechanical leverage)。

length1 + ratio * length2 == constant

举个例子,如果系数是2,那么 length1 的变化会是 length2 的两倍。另外连接 body1 的绳子的约束力将会是连接 body2 绳子的一半。



滑轮的一侧完全展开时,另一侧的绳子长度为零,这可能会出问题。此时,约束方程将变得奇异(糟糕)。因此,滑轮关节约束了每一侧的最大长度。另外出于游戏原因你可能也希望控制这个最大长度。最大长度能提高稳定性,以及提供更多的控制。

这是一个滑轮定义的例子:

b2Vec2 anchor1 = myBody1->GetWorldCenter();

b2Vec2 anchor2 =myBody2->GetWorldCenter();

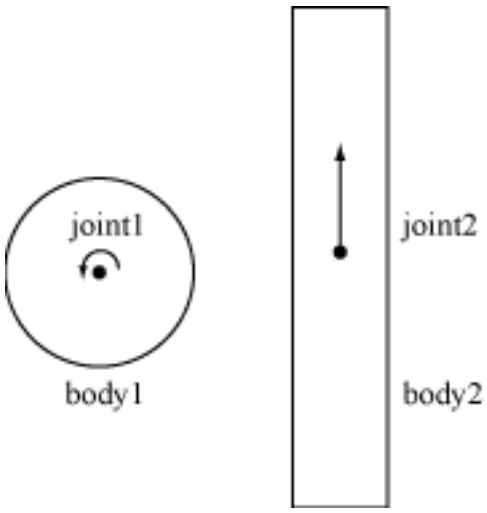
```
b2Vec2 groundAnchor1(p1.x, p1.y +10.0f);

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

float32 ratio = 1.0f;

b2PulleyJointDef jointDef;

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

jointDef.maxLength1 = 18.0f;

jointDef.maxLength2 = 20.0f;
```

滑轮关节提供函数得到当前长度。

```
float32 GetLength1() const;

float32 GetLength2() const;
```

8.9 齿轮关节(Gear Joint)

如果你想创建复杂的机械装置,可能需要齿轮。原则上,在 Box2D 中你可以用复杂的形状来模拟轮齿,但这并不十分高效,而且这样的工作可能有些乏味。另外,你还得小心地排列齿轮,保证轮齿能平稳地啮合。Box2D 提供了一个创建齿轮的更简单的方法:齿轮关节。



齿轮关节需要两个被旋转关节或移动关节接地(ground)的物体,你可以任意组合这些关节类型。另 外,创建旋转或移动关节时,Box2D需要地(ground)作为body1。

类似于滑轮的系数,你可以指定一个齿轮系数(ratio),齿轮系数可以为负。另外值得注意的是,当一 个是旋转关节(有角度的)而另一个是移动关节(平移)时,齿轮系数有长度单位，或者是长度单位的倒数。

coordinate1 + ratio * coordinate2 ==constant

这是一个齿轮关节的例子:

```
b2GearJointDef jointDef;

jointDef.body1 = myBody1;

jointDef.body2 = myBody2;

jointDef.joint1 = myRevoluteJoint;

jointDef.joint2 = myPrismaticJoint;
```

jointDef.ratio = 2.0f * b2_pi /myLength;

注意,齿轮关节依赖于两个其它关节,这是脆弱的:当其它关节被删除了会发生什么?

注意

齿轮关节总应该先于旋转或移动关节被删除。否则由于齿轮关节中的关节指针无效，你的代码将会因访问这些无效指针而导致崩溃。另外齿轮关节也应该在任何相关物体被删除之前删除。

8.10 鼠标关节(Mouse Joint)

在testbed例子中, 鼠标关节用于通过鼠标来操控物体。它试图将物体拖向当前鼠标光标的位置。而在旋转方面就没有限制。

鼠标关节的定义需要一个目标点(target point)，最大力(maximum force)，频率(frequency)，阻尼率(damping ratio)。目标点最开始与物体的锚点重合。最大力用于防止在多个动态物体相互作用时，会有激烈反应。你想将最大力设为多大就多大。频率和阻尼率用于创造一种弹性效果，就跟距离关节类似。

许多用户为了游戏的可玩性，会试图修改鼠标关节。用户常常希望鼠标关节有即时反应，精确的去到某个点。这情况下，鼠标关节表现并不好。你可以考虑一下用kinematic物体来替代。

8.11 线性关节(Line Joint)

线性关节跟移动关节(prismatic joint)类似，但没有旋转方面的约束。线性关节可以用来模拟悬挂着的车轮。更多细节请看b2LineJoint.h。

8.12 焊接关节(Weld Joint)

焊接关节的用途是使两个物体不能相对运动。看看testbed中的Cantilever例子，可以知道焊接关节有怎么样的表现。

用焊接关节来定义一个可分裂物体，这想法很诱人。但是，由于Box2D的迭代求解，关节焊得有点不稳。导致用焊接关节连接起来的物体会有所摆动。

创建可裂物体的更好方法是使用单个的物体，上面有很多fixture。当物体分裂时，你可以删掉原物体其中一个fixture，并重新创建一个新的物体，带有那个fixture。参考一下testbed中的Breakable例子。

9. Box2D v2.1.0用户手册翻译 - 第09章 接触(Contacts)

内容很多摘自

Aman JIANG(江超宇)翻译的Box2D v2.0.1 用户手册

第09章 接触(Contacts)

9.1 关于

接触(contact)是由 Box2D 创建的用于管理fixture间碰撞的对象。接触有不同的种类,它们都派生自b2Contact,用于管理不同类型形状之间的接触。例如, 有管理多边形之间碰撞的类,有管理圆形之间碰撞的类。

这是与接触有关的术语

接触点(contact point)
接触点就两个形状相互接触的点。在Box2D中，近似地认为在少数点处有接触。

接触法线(contact normal)
接触法线是一个单位向量，由一个fixture指向另一个fixture。按照惯例，向量由fixtureA指向fixtureB。

接触分隔(contact separation)
分隔正好与穿透(penetration)相反。当形状相重叠时, 分隔为负。有可能将来的Box2D版本中会以正隔离来创建触点, 所以当有触点的报告时你可能需要检查一下符号。

contact manifold

两个凸多边形相互接触，有可能会产生两个接触点。这些点都有相同的法线，所以就相它们分成一组，构成contact manifold，这是连续区域接触的一个惯例。

(译注: manifold不知道怎么翻译。我猜测manifold是指有相同特性的东西归成一类。接触点有相同的法线，就归成contact manifold。)

法向冲量(normal impulse)
法向力作用于接触点，用于防止形状相互穿透。为方便起见，Box2D使用冲量(impulses)。法向力与时间步相乘，构成法向冲量

切向冲量(tangent impulse)
切向力会在接触点生成，用于模拟摩擦。为方便起见，切向作用使用冲量的方式存储。

接触标识(contact ids)
从最开始的猜测值出发，Box2d得出当前时间步的触点压力，再重新利用当前时间步的压力结果去推测下一个时间步的压力结果。接触标识用于匹配跨越时间步的触点。标识包含了几何特征索引以便区分触点。

当两个fixture的AABB重叠时,接触就被创建了。有时碰撞筛选会阻止接触的创建。当AABB 不再重叠后接触会被摧毁。

也许你会皱起眉头,为了没有发生实际碰撞的形状(只是它们的 AABB)却创建了接触。好吧,的确是这样的,这是一个“ 鸡或蛋 ”的问题。我们并不知道是否需要一个接触,除非我们创建一个接触去分析碰撞。如果形状之间没有发生碰撞,我们需要正确地删除接触,或者,我们可以一直等到 AABB 不再重 叠 。为了提高性能, Box2D选择了后面这个方法。

9.2 接触类(Contact Class)

之前已经提及过，接触对象是Box2D内部创建和摧毁的，并不是由用户来创建。然而，你还是能够访问接触类并和它交互的。

你可以访问原始的contact manifold:

```
b2Manifold* GetManifold();

const b2Manifold* GetManifold()const;
```

你甚至可以修改manifold，一般情况下不提倡你怎样做。修改manifold是较高级的用法。

这个是帮助函数，去获取b2WorldManifold:

```
void GetWorldManifold(b2WorldManifold*worldManifold) const;
```

这使用了物体的当前位置去计算出接触点在world坐标下的位置。

传感器(Sensors)并不创建manifolds，所以要使用:

```
bool touching =sensorContact->IsTouching();
```

这函数对于非传感器(non-sensors)也有效。

从接触(contact)中你可以得到fixture, 从而再得到body。

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

b2Body* bodyA =fixtureA->GetBody();

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

你可以使一个接触失效。这仅仅在b2ContactListener::PreSolve事件中有效，下面会再进行讨论。

9.3 访问接触(Accessing Contacts)

你有几种方法来访问接触。为了访问接触，你可以直接查询world或者body结构，还可以实现一个接触监听器(contact listener)。

在world中，你可以遍历所有的接触:

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

{

        // process c

}
```

同样在body中，你也可以遍历所有接触。接触以图的方式存储，使用了接触边数据结构(contact edge structure),

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

{

        b2Contact* c = ce->contact;

        // process c

}
```

通过下面描述的接触监听器，你也可以访问接触。

注意

通过b2World或者b2Body直接访问，有可能会错过一些时间步中产生的临时接触。而使用 b2ContactListener 就可以很精确的得到全部结果。

9.4 接触监听器(Contact Listener)

通过实现 b2ContactListener 你就可以收到接触数据。接触监听器支持几种事件: 开始(begin)，结束(end), 求解前(pre-solve), 求解后(post-solve)。

```
class MyContactListener : publicb2ContactListener

{

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 }

};
```

注意

不要保存发送到b2ContactListener的指针。取而代之，用深拷贝的方式将触点数据保存到你自己的缓冲区。下面的例子演示了一种方法。

在运行期(run-time), 你可以创建listener的实例对象，并使用b2World::SetContactListener来注册这个对象。 但要保证当world对象存在时，listener要留在作用域中。

Begin事件
当两个fixture开始有重叠时，事件会被触发。传感器和非传感器都会触发这事件。这事件只能在时间步内(译注: 也就是b2World::step函数内部)发生。

End事件
当两个fixture不再重叠时，事件会被触发。传感器和非传感器都会触发这事件。当一个body被摧毁时，事件也有可能被触发。所以这事件也有可能发生在时间步之外。

Pre-Solve事件
在碰撞检测之后，但在碰撞求解之前，事件会被触发。这样可以给你一个机会，根据当前情况来决

定是否使这个接触失效。举个例子，在回调中使用b2Contact::SetEnabled(false)，你就可以实现单侧碰撞的功能。每次碰撞处理时，接触会重新生效，所以你在每一个时间步 中都应禁用那个接触。由于连续碰撞检测，pre-solve事件在单个时间步中有可能发生多次。

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

{

	b2WorldManifold worldManifold;

	contact->GetWorldManifold(&worldManifold);

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

	{

		contact->SetEnabled(false);

	}

}
```

如果要确认触点状态或得到碰撞速度，可以在pre-solve事件中处理。

```
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事件

当你可以得到碰撞冲量(collision impulse)的结果时，post-solve事件会发生。如果你不关心冲量，你可能只需要实现pre-solve事件。

在一个接触回调中去改变物理世界是诱人的。例如,你可能会以碰撞来施加伤害,并试图摧毁关联的角色和它的刚体。然而,Box2D并不允许你在回调中改变物理世界,因为你可能会摧毁 Box2D 正在运算的对象, 造成野指针。

处理触点的推荐方法是缓冲所有你关心的触点,并在时间步之后处理它们。一般在时间步之后你应该立即处理它们,否则其它客户端代码可能会改变物理世界,使你的缓冲失效。当你处理触点缓冲的时候,你可以去改变物理世界,但是你仍然应该小心不要造成无效的指针。在 testbed中有安全处理触点以避免无效指针的例子。

这是一小段 CollisionProcessing 测试中的代码,它演示了在操作触点缓冲时如何处理孤立物体。请注意注释。代码假定所有触点都缓冲于 b2ContactPoint 数组 m_points 中。

```
// 我们打算摧毁和contact有所关联的物体指针.

// 我们必须先缓存那些需要摧毁的物体，因为它们有可能被多个触点所共有。
```

```
const int32 k_maxNuke = 6;

b2Body* nuke[k_maxNuke];

int32 nukeCount = 0;



// 遍历contact缓存，摧毁相互接触时，无那么重的物体。

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;

                            }

                }

}



// 将nuke数组排序，使得重复的指针归在一起

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



// 删除body, 跳过重复的

int32 i = 0;

while (i < nukeCount)

{

                b2Body* b = nuke[i++];

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

```
        {

                ++i;

        }


        m_world->DestroyBody(b);

}
```

9.5 接触筛选(Contact Filtering)

通常,你不希望游戏中的所有物体都发生碰撞。例如,你可能会创建一个只有特定角色才能通过的门。这称之为接触筛选,因为一些交互被筛选出了。

通过实现b2ContactFilter类, Box2D允许定制接触筛选。这个类需要你实现一个ShouldCollide 函数, 这个函数接收两个b2Shape的指针作为参数。如果应该碰撞那么就返回true。

默认的ShouldCollide实现使用了“ 第06章， 夹具(Fixtures) ”定义的b2FilterData。

```
bool b2ContactFilter::ShouldCollide(b2Shape*shape1, b2Shape* shape2)

{

        const b2FilterData& filter1 =shape1->GetFilterData();
```

```
    const b2FilterData& filter2 =shape2->GetFilterData();



    if (filter1.groupIndex ==filter2.groupIndex &&

            filter1.groupIndex != 0)

    {

            return filter1.groupIndex> 0;

    }



    bool collide = (filter1.maskBits &filter2.categoryBits) != 0 &&

                                    (filter1.categoryBits& filter2.maskBits) != 0;

    return collide;

}
```

在运行期（run-time), 你可以创建自己的接触筛选实例，并使用b2World::SetContactFilter函数来注册。你要保证当world存在时，你的filter要保留在作用域中。

```
MyContactFilter filter;

world->SetContactFilter(&filter);

// filter留在作用域中
```

10. Box2D v2.1.0用户手册翻译 - 第10章 世界(World Class)

内容很多摘自

Aman JIANG(江超宇)翻译的Box2D v2.0.1 用户手册

第10章 世界(World Class)

关于
b2World类包含物体和关节。它管理着模拟的方方面面,并允许异步查询(就像AABB查询和光线投射)。 你与Box2D的大部分交互都将通过 b2World 对象来完成。

创建和摧毁world
创建一个world十分的简单。你只需提供一个重力矢量，和一个布尔量去指定物体是否可以休眠。 通常你会使用new和delete去创建和摧毁一个world。

```
b2World* myWorld = newb2World(gravity, doSleep);

... do stuff ...

delete myWorld;
```

使用World

world类含有用于创建和摧毁物体与关节的工厂函数, 已在物体和关节的章节中讨论过。 在这里我们讨论b2World的其它交互。

模拟(Simulation)
世界类用于驱动模拟。你需要指定一个时间步和一个速度及位置的迭代次数。例如:

```
float32 timeStep = 1.0f / 60.f;

int32 velocityIterations = 10;

int32 positionIterations = 8;

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

在时间步完成之后,你可以调查物体和关节的信息。最经常的情况是你会获取物体的位置,这样你才能更新你的角色并渲染它们。 你可以在游戏循环的任何地方执行时间步,但你应该注意事情发生的先后顺 序。例如,如果你想要在一帧(frame)中得到新物体的碰撞结果,你必须在时间步之前创建物体。

正如之前我在 HelloWorld 教程中说明的,你需要使用一个固定的时间步。使用大一些的时间步你可以在低帧率的情况下提升性能。 但通常情况下你应该使用一个不大于 1/30 秒 的时间步。1/60 的时间步通常会呈现一个高质量的模拟。

迭代次数控制了约束求解器会遍历多少次世界中的接触以及关节。更多的迭代总能产生更好的模拟, 但不要使用小频率大迭代数。60Hz和10次迭代远好于30Hz和20次迭代。

时间步之后, 你应该清除任何施加到物体之上的力。使用b2World::ClearForces可以完成。

```
myWorld->ClearForces();
```

探测世界(Exploring the World)
世界是物体和关节的容器。你可以获取世界中所有物体和关节并遍历它们。例如, 这段代码会唤醒世界中的所有物体:

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

{

        b->WakeUp();

}
```

不幸的是真实的程序可能很复杂。例如,下面的代码是有错误的:

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

{

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

        if (myActor->IsDead())

        {

                myWorld->DestroyBody(b);// 错误: 现在GetNext会返回无用信息(garbage)

        }

}
```

在物体摧毁之前一切都很顺利。一旦物体摧毁了，它的next指针就变得非法。所以 b2Body::GetNext() 就会返回无用信息。 解决方法是在物体摧毁之前拷贝next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);

	}

}
```

这能安全地摧毁当前物体。然而,你可能想要调用一个游戏的函数来摧毁多个物体,这时你需要十分小心。 解决方案取决于具体应用, 但为求方便， 在此我给出一种解决这问题的方法:

```
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();

            }

        }

}
```

很明显要保证这个能正确工作, GameCrazyBodyDestroyer对它都摧毁了什么必须要诚实。

AABB查询(AABB Queries)
有时你需要得出一个区域内的所有fixture。b2World类为此使用了broad-phase数据结构,提供了一个log(N) 的快速方法。你提供一个世界坐标的AABB和b2QueryCallback的一个实现。只要fixture的AABB和需查询的AABB有重合，world类就会调用你的b2QueryCallback类。返回true表示要继续查询，否则就返回false。例如，下面的代码找到所有大致与指定AABB相交的fixtures并唤醒所有关联的物体。

```
class MyQueryCallback : publicb2QueryCallback

{

public:

        bool ReportFixture(b2Fixture*fixture)

        {

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

```
            body->WakeUp();



            // 返回true，继续查询

            return true;

        }

};



...



MyQueryCallback callback;

b2AABB aabb;

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

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

myWorld->Query(&callback,aabb);

你不能假定回调函数会以固定的顺序执行。

光线投射(Ray Casts)
你可以使用光线投射去做现场(line-of-site)检查, 开枪扫射等等。通过实现一个回调类，并提供一个开始点和结束点，你就可以执行光线投射。只要fixture被光线穿过，world就会调用你提供的类。回调时会传递fixture，交点，单位法向量，和光线通过的分数距离(fractional distance along the ray)。你不能假定回调会以固定的顺序执行。

通过返回fraction， 你可以控制光线投射是否继续执行。返回的fraction为0，表示应该结束光线投射。fraction为1，表示投射应该继续执行，并且没有和其它形状相交。如果你返回参数列表中传近来的fraction, 表示光线会被裁剪到当前的和形状的相交点。这样通过返回适当的fraction值，你可以投射任何形状，投射所有形状，或者只投射最接近的形状。

另外你可以返回fraction为-1，去过虑fixture。这样光线投射会继续执行，并表现得似乎fixture根本就存在。

(译注:关于fraction的含义，可以看第04章的注释)

这里是个例子:

```
// This class captures the closesthit shape.

class MyRayCastCallback : publicb2RayCastCallback

{

public:

        MyRayCastCallback()

        {
```

```
        m_fixture = NULL;

}



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

                                constb2Vec2& 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);
```

注意

由于舍入误差，光线投射可能会通过在静态环境中的多边形之间的细小裂缝。如果这不是您的应用程序中的可接受的，请稍微扩大您的多边形。

力和冲量(Forces and Impulses)
你可以将力，扭矩，及冲量应用到物体上。当应用一个力或者冲量时，你需要提供一个在世界坐标下的受力点。这经常导致相对于质心，会有个扭矩。

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

void ApplyTorque(float32 torque);

void ApplyLinearImpulse(constb2Vec2& impulse, const b2Vec2& point);

void ApplyAngularImpulse(float32impulse);
```

应用力,扭矩或冲量会唤醒物体。有时这是不合需求的。例如,你可能想要应用一个固定的力,并允许物体休眠来提升性能。这时,你可以使用这样的代码:

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

{

        myBody->ApplyForce(myForce,myPoint);

}
```

坐标转换(Coordinate Transformations)
body类包含一些工具函数,它们可以帮助你在局部和世界坐标系之间转换点和向量。如果你不了解这些概念,请看 Jim Van Verth 和 Lars Bishop 的“ 游戏和交互应用的数学基础(Essential Mathematics for Games and Interactive Applications) ”。这些函数都很高效(当inline时)。

```
b2Vec2 GetWorldPoint(constb2Vec2& localPoint);

b2Vec2 GetWorldVector(constb2Vec2& localVector);

b2Vec2 GetLocalPoint(constb2Vec2& worldPoint);

b2Vec2 GetLocalVector(constb2Vec2& worldVector);
```

列表(Lists)
你可以遍历一个物体的fixture, **主要用途是帮助你访问**fixture**中的用户数据**。

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

{

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

        ... do something with data ...

}
```

你也可以用类似的方法遍历物体的关节列表。

body**也提供了访问相关**contact**的列表。你可以用来得到当前**contact**的信息。但使用时请小心，因为前一个时间步存在的**contact,**可能并不包含在当前列表中**。

11. Box2D v2.1.0用户手册翻译 - 第11章 杂项(Loose Ends)

内容很多摘自

Aman JIANG(江超宇)翻译的Box2D v2.0.1 用户手册

第11章 杂项(Loose Ends)

11.1 隐式摧毁

Box2D没有使用引用计数。你摧毁了body后，它就确实不存在了。访问指向已摧毁body的指针，会导致未定义的行为。 也就是说，你的程序可能会崩溃。以debug方式编译出的程序，Box2D的内存管理器会将已被摧毁实体占用的内存，都填上FDFDFDFD。一些时候， 这样可以使你更容易的找到问题的所在，进而修复问题。

如果你摧毁了Box2D实体, 你要确保所有指向这实体的引用都被移除。如果只有实体的单个引用，处理起来就很简单了。但如果有多个引用，你需要考虑是否去实现一个句柄类(handle class)，将原始指针封装起来。

当你使用Box2D时，会很经常创建并摧毁很多的物体(bodies), 形状(shapes),关节(joints)。某种程度上，这些实体是由Box2D自动管理的。当你摧毁了一个body,所有跟它有关联的形状，关节，接触都会自动被摧毁。这称为隐式摧毁。任何与这些关节或接触有连接的body也会被唤醒。这样处理通常也比较方便的。但是，你应该要注意一个关键问题:

注意

当body被摧毁时，所有依附其上的形状，关节也会被自动摧毁。你应该将任何指向这些形状和关节的指针清零。否则之后你试图访问或者再次摧毁这些形状或关节，你的程序会死得很惨。

大多数情况下，隐式摧毁还是很方便的，但也很容易使你的程序崩溃。你可能会将指向shape或关节的指针保存起来，当有关联的body摧毁时，这些指针就变得无效了。关节由某小段代码创建，有时你会认为这关节跟其它body都没有关系，但这只会使事情变得更糟糕。比如，testbed就创建了一个b2MouseJoint用来操作屏幕上的body。

Box2D提供了一种回调机制，当隐式摧毁发生时，你的程序可以收到通知，从而将无效指针清零。

你可以实现一个b2DestructionListener,这样当一个形状或关节隐式摧毁时，b2World 就能通知你。

```
class MyDestructionListener : publicb2DestructionListener

{

        void SayGoodbye(b2Joint* joint)

        {

                // remove all references tojoint.

        }

};
```

你可以将摧毁监听器(destruction listener)注册到world对象中。在world对象初始化时，你就应该这样做了。

```
myWorld->SetListener(myDestructionListener);
```

12. Box2D v2.1.0用户手册翻译 - 第12, 13, 14章

内容很多摘自

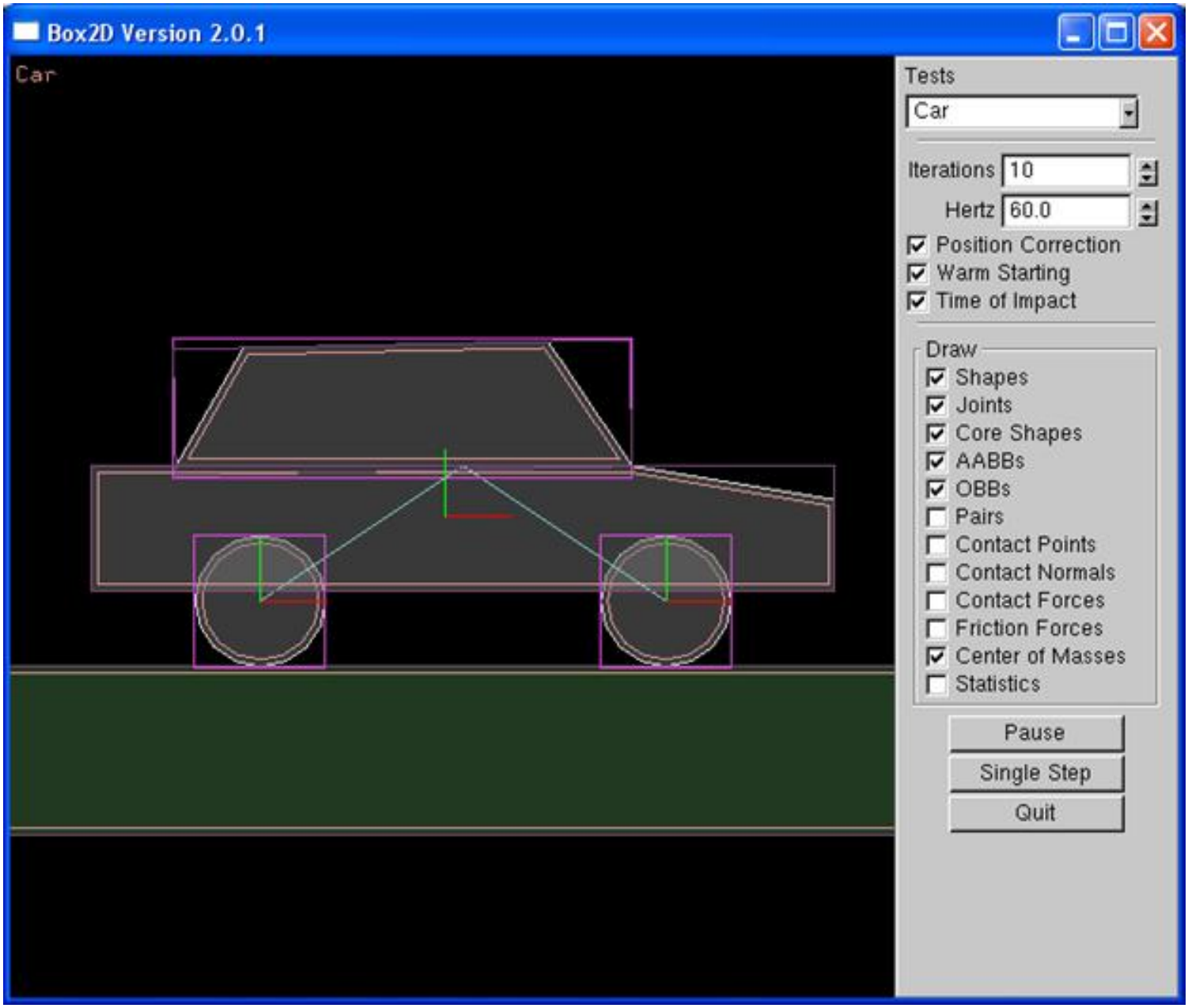
Aman JIANG(江超宇)翻译的Box2D v2.0.1 用户手册

第12章 调试绘图(Debug Drawing)

实现 b2DebugDraw 可得到物理世界的细部图,这里是可用的实体:

- 形状轮廓
- 关节连通性
- 核心形状(用于连续碰撞)
- broad-phase的AABB,包括世界的AABB
- polygon oriented bounding boxes (OBBs)
- broad-phase pairs (potential contacts)
- 质心

这是绘制这些物理实体的首先方法，比直接访问数据要好。因为很多的必要信息只能在内部访问并时有变更。



testbed使用debug drawing和接触监听器来绘制物理实体。它本身就是个好例子，演示了怎样去实现debug drawing以及怎样会绘制接触点。

第13章 限制(Limitations)

Box2D使用了一些数值近似来让模拟更高效。这就带来一些限制。

这是当前的限制:

1. 将重的物体放到相对很轻的物体上面，会不稳定。当质量比到10:1时，稳定性就会降低。

2. Polygons may not slide smoothly over chains of edge shapes or other polygon shapes that are aligned. For this reason, tile-based environments may not have smooth collision with box-like characters.这个问题在将来会被修复。

3. 用关节将body链接起来，如果是较轻的body吊着较重的物体，body链接有可能被拉伸。比如，一条很轻的锁链吊着个很重的球，就可能不稳定。当质量比超过10:1时，稳定性就会降低。

4. 通常还有约0.5cm的间隙，就检测到形状与形状碰撞。

5. 连续碰撞是按顺序处理的。在发生撞击事件时，body会向后移动，并在剩余的时间步内停留在那里。这可能会使得快速移动的物体，移动起来不太平滑。

第14章 参考(References)

Erin Catto 的 GDC 教程: http://code.google.com/p/box2d/downloads/list

3D交互环境下的碰撞检测(Collision Detection in Interactive 3D Environments), Gino van den Bergen, 2004

实时碰撞检测(Real-Time Collision Detection), Christer Ericson, 2005