Box2D v2.1.0 User Manual

Box2D v2.0.1 用户手册

原文：Box2D v2.0.2 User Manual

译者：Aman JIANG(江超宇)，翻译信息。

1. 导言

1.1 关于

Box2D 是一个用于游戏的 2D 刚体仿真库。程序员可以在他们的游戏里使用它，它可以使物体的运动更加可信，让世界看起来更具交互性。从游戏的视角来看，物理引擎就是一个程序性动画(procedural animation)的系统，而不是由动画师去移动你的物体。你可以让牛顿来做导演。

Box2D 是用可移植的 C++ 来写成的。引擎中定义的大部分类型都有 *b2* 前缀，希望这能消除它和你游戏引擎之间的名字冲突。

1.2 必备条件

在此，我假定你已经熟悉了基本的物理学概念，例如质量，力，扭矩和冲量。如果没有，请先考虑读一下 Chris Hecker 和 David Baraff (google 这些名字)的那些教程，你不需要了解得非常细致，但他们可以使你很好地了解一些基本概念，以便你使用 Box2D。

Wikipedia 也是一个极好的物理和数学知识的获取源，在某些方面它可能比 google 更有用，因为它的内容经过了精心的整理。

这不是必要的，但如果你好奇 Box2D 内部是如何工作的，你可以看 这些文档。

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

1.3 核心概念

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

刚体(rigid body)

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

形状(shape)

一块严格依附于物体(body)的 2D 碰撞几何结构(collision geometry)。形状具有摩擦(friction)和恢复(restitution)的材料性质。

约束(constraint)

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

接触约束(contact constraint)

一个防止刚体穿透，以及用于模拟摩擦(friction)和恢复(restitution)的特殊约束。你永远都不必创建一个接触约束，它们会自动被 Box2D 创建。

关节(joint)

它是一种用于把两个或多个物体固定到一起的约束。Box2D 支持的关节类型有：旋转，棱柱，距离等等。关节可以支持限制(limits)和马达(motors)。

关节限制(joint limit)

一个关节限制(joint limit)限定了一个关节的运动范围。例如人类的胳膊肘只能做某一范围角度的运动。

关节马达(joint motor)

一个关节马达能依照关节的自由度来驱动所连接的物体。例如，你可以使用一个马达来驱动一个肘的旋转。

世界(world)

一个物理世界就是物体，形状和约束相互作用的集合。Box2D 支持创建多个世界，但这通常是不必要的。

2. Hello Box2D

2.1 创建一个世界

每个 Box2D 程序都将从一个世界对象(world object)的创建开始。这是一个管理内存，对象和模拟的中心。

要创建一个世界对象，我们首先需要定义一个世界的包围盒。Box2D 使用包围盒来加速碰撞检测。尺寸并不关键，但合适的尺寸有助于性能。这个包围盒过大总比过小好。

```
b2AABB worldAABB;
worldAABB.lowerBound.Set(-100.0f, -100.0f);
worldAABB.upperBound.Set(100.0f, 100.0f);
```

- *注意：*worldAABB 应该永远比物体所在的区域要大，让 worldAABB 更大总比太小要好。如果一个物体到达了 worldAABB 的边界，它就会被冻结并停止模拟。

接下来我们定义重力矢量。是的，你可以使重力朝向侧面(或者你只好转动你的显示器)。并且，我们告诉世界(world)当物体停止移动时允许物体休眠。一个休眠中的物体不需要任何模拟。

```
b2Vec2 gravity(0.0f, -10.0f);
bool doSleep = true;
```

现在我们创建世界对象。通常你需要在堆(heap)上创建世界对象，并把它的指针保存在某一结构中。然而，在这个例子中也可以在栈上创建。

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

那么现在我们有了自己的物理世界，让我们再加些东西进去。

2.2 创建一个地面盒

物体通常由以下步骤来创建：

1. 使用位置(position)，阻尼(damping)等定义一个物体
2. 使用世界对象创建物体
3. 使用几何结构，摩擦，密度等定义形状
4. 在物体上创建形状
5. 可选地调整物体的质量以和附加的形状相匹配

第一步，我们创建地面体。要创建它我们需要一个*物体定义*(body definition)，通过物体定义我们来指定地面体的初始位置。

```
b2BodyDef groundBodyDef;
groundBodyDef.position.Set(0.0f, -10.0f);
```

第二步，将物体定义传给世界对象来创建地面体。世界对象并不保存到物体定义的引用。地面体是作为静态物体(static body)创建的，静态物体之间并没有碰撞，它们是固定的。当一个物体具有零质量的时候 Box2D 就会确定它为静态物体，物体的默认质量是零，所以它们默认就是静态的。

```
b2Body* ground = world.CreateBody(&groundBodyDef);
```

第三步，我们创建一个地面的多边形定义。我们使用 SetAsBox 简捷地把地面多边形规定为一个盒子(矩形)形状，盒子的中点就位于父物体的原点上。

```
b2PolygonDef groundShapeDef;
groundShapeDef.SetAsBox(50.0f, 10.0f);
```

其中，SetAsBox 函数接收了半个宽度和半个高度，这样的话，地面盒就是 100 个单位宽(x 轴)以及 20 个单位高(y 轴)。Box2D 已被调谐使用米，千克和秒来作单位，所以你可以用米来考虑长度。然而，改变单位系统是可能的，随后的文档中会有讨论。

在第四步中，我们在地面体上创建地面多边形，以完成地面体。

```
groundBody->CreateShape(&groundShapeDef);
```

重申一次，Box2D 并不保存到形状或物体的引用。它把数据拷贝到 b2Body 结构中。

注意每个形状都必须有一个父物体，即使形状是静态的。然而你可以把所有静态形状都依附于单个静态物体之上。这个静态物体之需求是为了保证 Box2D 内部的代码更具一致性，以减少潜在的 bug 数

量。

可能你已经注意到了，大部分 Box2D 类型都有一个 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 创建一个动态物体

现在我们已经有了一个地面体，我们可以使用同样的方法来创建一个动态物体。除了尺寸之外的主要区别是，我们必须为动态物体设置质量性质。

首先我们用 CreateBody 创建物体。

```
b2BodyDef bodyDef;
bodyDef.position.Set(0.0f, 4.0f);
b2Body* body = world.CreateBody(&bodyDef);
```

接下来我们创建并添加一个多边形形状到物体上。注意我们把密度设置为 1，默认的密度是 0。并且，形状的摩擦设置到了 0.3。形状添加好以后，我们就使用 SetMassFromShapes 方法来命令物体通过形状去计算其自身的质量。这暗示了你可以给单个物体添加一个以上的形状。如果质量计算结果为 0，那么物体会变成真正的静态。物体默认的质量就是零，这就是为什么我们无需为地面体调用 SetMassFromShapes 的原因。

```
b2PolygonDef shapeDef;
shapeDef.SetAsBox(1.0f, 1.0f);
shapeDef.density = 1.0f;
shapeDef.friction = 0.3f;
body->CreateShape(&shapeDef);
body->SetMassFromShapes();
```

这就是初始化过程。现在我们已经准备好开始模拟了。

2.4 模拟(Box2D 的)世界

我们已经初始化好了地面盒和一个动态盒。现在是让牛顿接手的时刻了。我们只有少数几个问题需要考虑。

Box2D 中有一些数学代码构成的*积分器*(integrator)，积分器在离散的时间点上模拟物理方程，它将与游戏动画循环一同运行。所以我们需要为 Box2D 选取一个时间步，通常来说游戏物理引擎需要至少 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;
```

除了积分器之外，Box2D 中还有*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，我们需要迭代所有约束多次。建议的 Box2D 迭代次数是 10 次。你可以按自己的喜好去调整这个数，但要记得它是速度与质量之间的平衡。更少的迭代会增加性能并降低精度，同样地，更多的迭代会减少性能但提高模拟质量。这是我们选择的迭代次数：

```
int32 iterations = 10;
```

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

现在我们可以开始模拟循环了，在游戏中模拟循环应该并入游戏循环。每次循环你都应该调用 b2World::Step，通常调用一次就够了，这取决于帧频以及物理时间步。

这个 Hello World 程序设计得非常简单，所以它没有图形输出。胜于完全没有输出，代码会打印出动态物体的位置以及旋转角度。Yay！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, iterations);
    b2Vec2 position = body->GetPosition();
    float32 angle = body->GetAngle();
    printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle);
}
```

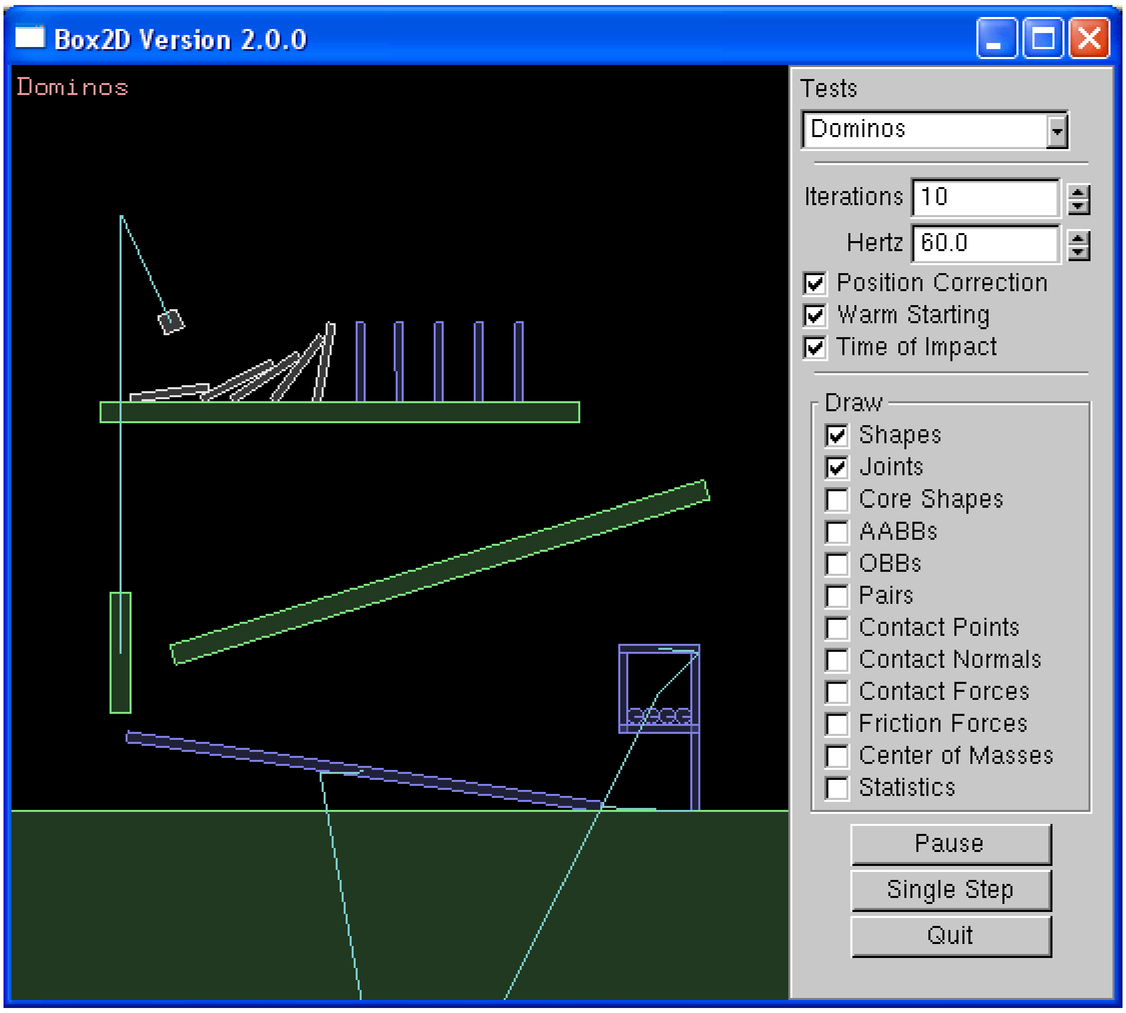
2.5 清理工作

当一个世界对象超出它的作用域，或通过指针将其 delete 时，所有物体和关节的内存都会被释放。这能使你的生活变得更简单。然而，你应该将物体，形状或关节的指针都清零，因为它们已经无效了。

2.6 关于 Testbed

一旦你征服了 HelloWorld 例子，你应该开始看 Box2D 的 testbed 了。testbed 是一个单元测试框架以及演示环境，这是一些它的特点：

- 可移动和缩放的摄像机
- 鼠标拣选动态物体的形状
- 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



Tests: Dominos
Iterations: 10
Hertz: 60.0
[✓] Position Correction
[✓] Warm Starting
[✓] Time of Impact

Draw
[✓] Shapes
[✓] Joints
[ ] Core Shapes
[ ] AABBs
[ ] OBBs
[ ] Pairs
[ ] Contact Points
[ ] Contact Normals
[ ] Contact Forces
[ ] Friction Forces
[ ] Center of Masses
[ ] Statistics

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 本身对于渲染是无知的，就像 HelloWorld 例子一样，使用 Box2D 并不一定需要渲染。

3. API 设计

3.1 内存管理

Box2D 的许多设计决策都是为了能快速有效地使用内存。在本节我将论述 Box2D 如何和为什么要分配内存。

Box2D 倾向于分配大量的小对象(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，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分配器(allocator)。

Box2D 的解决方案是使用*小型对象*分配器(SOA)，SOA 维护了许多不定尺寸的可生长的池(growable pool)。当有内存分配请求时，SOA 会返回一块最匹配的内存。当内存块释放掉以后，它会回到池中。这些操作都十分快速，导致很小的堆流量。

因为 Box2D 使用了 SOA，所以你应该永远也不必去 new 或 malloc 物体，形状或关节。你只需分配一个 b2World，它为你提供了创建物体，形状和关节的工厂(factory)。这使得 Box2D 可以使用 SOA 并且将赤裸的细节隐藏起来。永远也不要去 delete 或 free 一个物体，形状或关节。

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(stack)分配器来消除单步堆分配。你不需要关心栈分配器，但在此作一个了解还是不错的。

3.2 工厂和定义

如上所述，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 的时候，你需要调用 b2World 的工厂函数。

这些是创建函数：

```
b2Body* b2World::CreateBody(const b2BodyDef* def)
b2Joint* b2World::CreateJoint(const b2JointDef* def)
```

这是对应的摧毁函数：

```
void b2World::DestroyBody(b2Body* body)
void b2World::DestroyJoint(b2Joint* joint)
```

当你创建一个物体或关节的时候，你需要提供一个定义(definition，简写为 def)。这些定义包含了创建物体或关节的所有相关信息。通过这样的方法，我们就能预防构造错误，使函数参数的数量较少，提供有意义的默认参数，并减少访问子(accessor)的数量。

因为形状必须有父物体，所以 b2Body 上有创建和摧毁形状的工厂：

```
b2Shape* b2Body::CreateShape(const b2ShapeDef* def)
void b2Body::DestroyShape(b2Shape* shape)
```

工厂并不保留到定义的引用，所以你可以在栈上创建定义，临时的保存它们。

3.3 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。

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。想象一下使用一个被调谐好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。移动物体的尺寸大约应该保持在 0.1 到 10 米之间。你可能需要一些缩放系统来渲染你的场景和物体。Box2D 中的例子是使用 OpenGL 的视口来变换的。

3.4 用户数据

b2Shape，b2Body 和 b2Joint 类都允许你通过一个 void 指针来附加用户数据。这在你测试 Box2D 数据结构，以及你想把它们联系到自己的引擎中的时候是较方便的。

举个典型的例子，在角色上的刚体中附加到角色的指针，这就构成了一个循环引用。如果你有角色，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);
```

这是一些需要用户数据的案例：

- 使用碰撞结果给角色施加伤害
- 当玩家进入一个包围盒时播放一段脚本事件
- 当 Box2D 通知你一个关节即将摧毁时访问一个游戏结构

记得用户数据是可选的，并且能放入任何东西。然而，你需要保持一致性。例如，如果你想在一个物体中保存一个角色的指针，那你就应该在所有物体中都保存一个角色指针。不要在一个物体中保存角色指针，却在另一个物体中保存一个其它指针。这可能会导致程序崩溃。

3.5 C++ 相关面

C++ 有着强大的封装和多态，但在 API 设计方面却不那么强大。在创建一个 C++ 库的时候总会存在许多有意义的取舍。

我们是否应该使用抽象工厂或 *pimpl* 模式？它们能使 API 看起来更简洁，但它们最终会妨碍调试和高效开发。

我们是否有必要使用私有数据和友元(friend)？也许，但最后友元的数量可能会变得荒谬。

我们是否应该用一个 C-API 封装 C++ 代码？也许，但这是额外的工作，并且可能会导致非最佳的内部选择。另外，C-API 也难于调试和维护，一个 C-API 同时也破坏了封装。

我为 Box2D 选择了最容易的方法。有时候一个类可以包含其设计和函数，所以我使用公有函数和私有数据。其它情况下我使用了全部公有的成员的类和结构。这样的选择使我能快速地开发代码，很容易调试，并且当维护紧密的封装时最小化了内部混乱。如此，你并不能看见一个简单干净的 API。当然，你拥有的这个漂亮的手册能帮助你摆脱困扰 :)

3.6 稻草人

如果你不喜欢这个 API 的设计，that's ok！你拥有源代码！诚挚地，如果你有任何关于 Box2D 的反馈，请在 论坛 里留下意见。

4. 世界

4.1 关于

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

4.2 创建和摧毁一个世界

创建一个世界十分的简单。你只需提供一个包围盒和一个重力向量。

轴对齐包围盒(AABB)应该包围着世界。稍微比世界大一些的包围盒可以提升性能，比方 2 倍大小才安全。如果你的许多物体都掉进了深渊，你应该侦测并移除它们。这能提升性能并预防浮点溢出。

要创建或摧毁一个世界你需要使用 new 和 delete：

```
b2World* myWorld = new b2World(aabb, gravity, doSleep);
// ... do stuff ...
delete myWorld;
```

• *注意：*请回忆，AABB 的世界应该比你物体所在的区域要大。如果物体离开了 AABB，它们将被冻结。这不是一个 bug。

4.3 使用一个世界

世界类包含着用于创建和摧毁物体与关节的工厂，这些工厂会在后面的物体和关节的章节中讨论。在此我们讨论一些 b2World 的其它交互。

4.3.1 模拟

世界类用于驱动模拟。你需要指定一个时间步和一个迭代次数。例如：

```
float32 timeStep = 1.0f / 60.f;
int32 iterationCount = 10;
myWorld->Step(timeStep, iterationCount);
```

在时间步完成之后，你可以调查物体和关节的信息。最可能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，但你应该意识到事情发生的顺序。例如，如果你想要在一帧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。

4.3.2 扫描世界

如上所述，世界就是一个物体和关节的容器。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();
}
```

不幸的是生活有时很复杂。例如，下面的代码是错误的：

```
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.
    }
}
```

在一个物体摧毁之前一切都很顺利。一旦一个物体摧毁了，它的 next 指针就变得非法，所以 b2Body::GetNext() 就会返回垃圾。解决方法是在摧毁之前拷贝 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，但在此我给出一种方法：

```
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。

4.3.3 AABB 查询

有时你需要求出一个区域内的所有形状。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，而 b2World 会返回一个所有大概相交于此 AABB 的形状之数组。这不是精确的，因为函数实际上返回那些 AABB 与规定之 AABB 相交的形状。例如，下面的代码找到所有大概与指定 AABB 相交的形状并唤醒所有关联的物体。

```
b2AABB aabb;
aabb.minVertex.Set(-1.0f, -1.0f);
aabb.maxVertex.Set(1.0f, 1.0f);
const int32 k_bufferSize = 10;
b2Shape *buffer[k_bufferSize];
int32 count = myWorld->Query(aabb, buffer, k_bufferSize);
for (int32 i = 0; i < count; ++i)
{
    buffer[i]->GetBody()->WakeUp();
}
```

5. 物体

5.1 关于

物体具有位置和速度。你可以应用力，扭矩和冲量到物体。物体可以是静态的或动态的，静态物体永远不会移动，并且不会与其它静态物体发生碰撞。

物体是形状的主干，物体携带形状在世界中运动。在 Box2D 中物体总是刚体，这意味着同一刚体上的两个形状永远不会相对移动。

通常你会保存所有你所创建的物体的指针，这样你就能查询物体的位置，并在图形实体中更新它的位置。另外在不需要它们的时候你也需要通过它们的指针摧毁它们。

5.2 物体定义

在创建物体之前你需要创建一个物体定义(b2BodyDef)。你可以把物体定义创建在栈上，也可以在你的游戏数据结构中保存它们。这取决于你的选择。

Box2D 会从物体定义中拷贝出数据，它不会保存到物体定义的指针。这意味着你可以循环使用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。

5.2.1 质量性质

有多种建立物体质量性质的方法：

1. 在物体定义中显式地设置

2. 显式地在物体上设置(在其创建之后)

3. 基于物体上形状之密度设置

在很多游戏环境中，根据形状密度计算质量是有意义的。这能帮助确保物体有合理和一致的质量。

然而，其它游戏环境可能需要指定质量值。例如，可能你有一个机械装置，需要一个精确的质量。

你可以这样在物体定义中显式地设置质量性质：

```
bodyDef.massData.mass = 2.0f;   // the body's mass in kg
bodyDef.center.SetZero();       // the center of mass in local coordinates
bodyDef.I = 3.0f;               // the rotational inertia in kg*m^2.
```

其它设置质量性质的方法在本文档其它部分有描述。

5.2.2 位置和角度

物体定义为你提供了一个在创建时初始化位置的机会，这要比在世界原点创建物体而后移动它到某个位置更具性能。

一个物体上主要有两个令人感兴趣的点。其中一个是物体的原点，形状和关节都相对于物体的原点而被附加。另一个点是物体的质心。质心取决于物体上形状的质量分配，或显式地由 b2MassData 设置。Box2D 内部的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，因此你会指定物体的原点。你可能也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性质，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);   // the body's origin position.
bodyDef.angle = 0.25f * b2_pi;      // the body's angle in radians.
```

5.2.3 阻尼

阻尼用于减小物体在世界中的速率。阻尼与摩擦是不同的，因为摩擦仅在物体有接触的时候才会发生，而阻尼的模拟要比摩擦便宜多了。然而，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 到无穷之间，0 的就是没有阻尼，无穷就是满阻尼。通常来说，阻尼的值应该在 0 到 0.1 之间，我通常不使用线性阻尼，因为它会使物体看起来发飘。

```
bodyDef.linearDamping = 0.0f;
bodyDef.angularDamping = 0.01f;
```

阻尼相似于稳定性与性能，阻尼值较小的时候阻尼效应几乎不依赖于时间步，而阻尼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。

5.2.4 休眠参数

休眠是什么意思？好的。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。当一个物体停止了运动时，我们喜欢停止去模拟它。

当 Box2D 确定一个物体(或一组物体)已经停止移动时，物体就会进入休眠状态，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来。当物体上的关节或触点被摧毁的时候，它们同样会醒来。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.isSleeping = false;
```

5.2.5 子弹

有的时候，在一个时间步内可能会有大量的刚体同时运动。如果一个物理引擎没有处理好大幅度运动的问题，你就可能会看见一些物体错误地穿过了彼此。这种效果被称为隧道效应(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，这是通过从形状的旧位置到新位置的扫描来完成的。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。如果有必要这个步骤会重复执行。

一般 CCD 不会应用于动态物体之间，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，譬如，你可能想用一颗高速的子弹去射击薄壁。没有 CCD，子弹就可能会隧穿薄壁。

高速移动的物体在 Box2D 被称为子弹(bullet)，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.isBullet = true;
```

子弹开关只影响动态物体。

CCD 的成本是昂贵的，所以你可能不希望所有运动物体都成为子弹。所以 Box2D 默认只在动态物体和静态物体之间使用 CCD，这是防止物体逃脱游戏世界的一个有效方法。然而，可能你有一些高速移动的物体需要一直使用 CCD。

5.3 物体工厂

物体的创建和摧毁是由世界类提供的物体工厂来完成的。这使得世界可以通过一个高效的分配器来创建物体，并且把物体加入到世界数据结构中。

物体可以是动态或静态的，这取决于质量性质。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。

静态物体不会受其它物体的作用而移动。你可以手动地移动静态物体，但你必须小心，不要挤压到静态物体之间的动态物体。另外，当你移动静态物体的时候，摩擦不会正确工作。在一个静态物体上附加数个形状，要比在多个静态物体上附加单个形状有更好的性能。在内部，Box2D 会设置静态物体的质

量，并把质量反转为零，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，并复用同样的物体定义。

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，你必须小心地处理那些已失效的物体指针。

5.4 使用物体

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

5.4.1 质量数据

你可以在运行时调整一个物体的质量，这通常是在添加或移除物体上之形状时完成的。可能你会根据物体上的当前形状来调整其质量。

```
void SetMassFromShapes();
```

可能你也会直接设置质量。例如，你可能会改变形状，但你只想使用自己的质量公式。

```
void SetMass(const b2MassData* massData);
```

通过以下这些函数可以获得物体的质量数据：

```
float32 GetMass() const;
float32 GetInertia() const;
const b2Vec2& GetLocalCenter() const;
```

5.4.2 状态信息

物体的状态含有多个方面，通过这些函数你可以访问这些状态数据：

```
bool IsBullet() const;
void SetBullet(bool flag);

bool IsStatic() const;
bool IsDynamic() const;

bool IsFrozen() const;

bool IsSleeping() const;
void AllowSleeping(bool flag);
void WakeUp();
```

The bullet state is described in Section 5.2.5, “Bullets”. The frozen state is described in Section 9.1, “World Boundary”.

其中，子弹状态在 5.2.5 子弹 中有描述，冻结状态在 9.1 世界边界 中有描述。

5.4.3 位置和速度

你可以访问一个物体的位置和角度，这在你渲染相关游戏角色时很常用。你也可以设置位置，尽管这不怎么常用。

```
bool SetXForm(const b2Vec2& position, float32 angle);
const b2XForm& GetXForm() const;
const b2Vec2& GetPosition() const;
float32 GetAngle() const;
```

你可以访问世界坐标的质心。许多 Box2D 内部的模拟都使用质心，然而，通常你不必访问它。取而代之，你一般应该关心物体变换。

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

你可以访问线速度与角速度，线速度是对于质心所言的。

```
void SetLinearVelocity(const b2Vec2& v);
b2Vec2 GetLinearVelocity() const;
void SetAngularVelocity(float32 omega);
float32 GetAngularVelocity() const;
```

5.4.4 力和冲量

你可以对一个物体应用力，扭矩，以及冲量。当应用一个力或冲量时，你需要提供一个世界位置。这常常会导致对质心的一个扭矩。

```
void ApplyForce(const b2Vec2& force, const b2Vec2& point);
void ApplyTorque(float32 torque);
void ApplyImpulse(const b2Vec2& impulse, const b2Vec2& point);
```

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稳定的力，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->IsSleeping() == false)
{
    myBody->ApplyForce(myForce, myPoint);
}
```

5.4.5 坐标转换

物体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“Essential Mathematics for Games and Interactive Applications”。这些函数都很高效，所以可放心使用。

```
b2Vec2 GetWorldPoint(const b2Vec2& localPoint);
b2Vec2 GetWorldVector(const b2Vec2& localVector);
b2Vec2 GetLocalPoint(const b2Vec2& worldPoint);
b2Vec2 GetLocalVector(const b2Vec2& worldVector);
```

5.4.6 列表

你可以遍历一个物体的形状，其主要用途是帮助你访问形状之用户数据。

```
for (b2Shape* s = body->GetShapeList(); s; s = s->GetNext())
{
    MyShapeData* data = (MyShapeData*)s->GetUserData();
    ... do something with data ...
}
```

你也可以用类似的方法遍历物体的关节列表。

6 形状

6.1 关于

形状就是物体上的碰撞几何结构。另外形状也用于定义物体的质量。也就是说，你来指定密度，Box2D 可以帮你计算出质量。

形状具有摩擦和恢复的性质。形状还可以携带筛选信息，使你可以防止某些游戏对象之间的碰撞。

形状永远属于某物体，单个物体可以拥有多个形状。形状是抽象类，所以在 Box2D 中可以实现许多类型的形状。如果你有勇气，那便可以实现出自己的形状类型(和碰撞算法)。

6.2 形状定义

形状定义用于创建形状。通用的形状数据会保存在 b2ShapeDef 中，特殊的形状数据会保存在其派生类中。

6.2.1 摩擦和恢复

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，但使用相同的参数。摩擦在 Box2D 中会被正确地模拟，并且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，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);
```

恢复可以使对象弹起，想象一下，在桌面上方丢下一个小球。恢复的值通常设置在 0 到 1 之间，0 的意思是小球不会弹起，这称为*非弹性*碰撞；1 的意思是小球的速度会得到精确的反射，这称为*完全弹性*碰撞。恢复是通过这样的公式计算的：

```
float32 restitution;
restitution = b2Max(shape1->restitution, shape2->restitution);
```

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

6.2.2 密度

Box2D 可以根据附加形状的质量分配来计算物体的质量以及转动惯量。直接指定物体质量常常会导致不协调的模拟。因此，推荐的方法是使用 b2Body::SetMassFromShape 来根据形状设置质量。

6.2.3 筛选

碰撞筛选是一个防止某些形状发生碰撞的系统。譬如说，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 通过种群和组支持了这样的碰撞筛选。

Box2D 支持 16 个种群，对于任何一个形状你都可以指定它属于哪个种群。你还可以指定这个形状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：

```
playerShapeDef.filter.categoryBits = 0x0002;
monsterShapeDef.filter.categoryBits = 0x0004;
playerShape.filter.maskBits = 0x0004;
monsterShapeDef.filter.maskBits = 0x0002;
```

碰撞组可以让你指定一个整数的组索引。你可以让同一个组的所有形状总是相互碰撞(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，shape1 和 shape2 总是碰撞，而 shape3 和 shape4 永远不会碰撞。

```
shape1Def.filter.groupIndex = 2;
shape2Def.filter.groupIndex = 2;
shape3Def.filter.groupIndex = -8;
shape4Def.filter.groupIndex = -8;
```

不同组索引之间形状的碰撞会按照种群和掩码来筛选。换句话说，组筛选比种群筛选有更高的优选权。

注意在 Box2D 中的其它碰撞筛选，这里是一个列表：

- 静态物体上的形状永远不会与另一个静态物体上的形状发生碰撞
- 同一个物体上的形状之间永远不会发生碰撞
- 你可以有选择地启用或禁止由关节连接之物体上的形状之间是否碰撞

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 来存取已存在形状之 b2FilterData 结构。Box2D 会缓存筛选结果，所以你需要使用 b2World::Refilter 手动地进行重筛选。

6.2.4 传感器

有时候游戏逻辑需要判断两个形状是否相交，但却不应该有碰撞反应。这可以通过传感器 (sensor)来完成。传感器会侦测碰撞而不产生碰撞反应。

你可以将任一形状标记为传感器，传感器可以是静态或动态的。记得，每个物体上可以有多个形状，并且传感器和实体形状是可以混合的。

```
myShapeDef.isSensor = true;
```

6.2.5 圆形定义

b2CircleDef 扩充了 b2ShapeDef 并增加一个半径和一个局部位置。

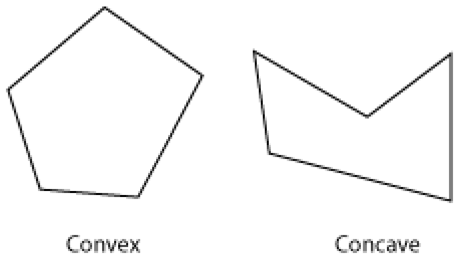
```
b2CircleDef def;
def.radius = 1.5f;
def.localPosition.Set(1.0f, 0.0f);
```

6.2.6 多边形定义

b2PolyDef 用于定义凸多边形。要正确地使用需要一点点技巧，所以请仔细阅读。最大顶点数由 b2_maxPolyVertices 定义，当前是 8。如果你需要更多顶点，你必须修改 b2Settings.h 中的 b2_maxPolyVertices。

当创建多边形定义时，你需要给出所用的顶点数目。这些顶点必须按照相对于右手坐标系之 z 轴*逆时针*(CCW)的顺序定义。在你的屏幕上可能是顺时针的，这取决于你的坐标系统规则。

多边形必须是*凸多边形*，也就是，每个顶点都必须指向外面。最后，你也不应该重叠任何顶点。Box2D 会自动地封闭环路。



这里是一个三角形的多边形定义的例子：

```
b2PolygonDef triangleDef;
triangleDef.vertexCount = 3;
triangleDef.vertices[0].Set(-1.0f, 0.0f);
triangleDef.vertices[1].Set(1.0f, 0.0f);
triangleDef.vertices[2].Set(0.0f, 2.0f);
```

顶点会被定义于父物体之坐标系中。如果你需要在物体内偏移多边形，那就偏移所有顶点。

为了方便，有些函数可以把多边形初始化为矩形。可以是轴对齐的、中心点位于物体原点的矩形，或者是有角度有偏移的矩形。

```
b2PolygonDef alignedBoxDef;
float32 hx = 1.0f; // half-width
float32 hy = 2.0f; // half-height
alignedBodyDef.SetAsBox(hx, hy);

b2PolygonDef orientedBoxDef;
b2Vec2 center(-1.5f, 0.0f);
float32 angle = 0.5f * b2_pi;
orientedBoxDef.SetAsBox(hx, hy, center, angle);
```

6.3 形状工厂

初始化一个形状定义，而后将其传递给父物体；形状就是这样创建的。

```
b2CircleDef circleDef;
circleDef.radius = 3.0f;
circleDef.density = 2.5f;
b2Shape* myShape = myBody->CreateShape(&circleDef);
// [optionally store shape pointer somewhere]
```

这样就创建了形状，并将其添加到了物体之上。你无须存储形状的指针，因为当父物体摧毁时形状也将自动地摧毁(请看 9.2 隐式摧毁)。

在添加了形状到物体之后，你可能需要根据形状重新计算物体的质量性质。

```
myBody->SetMassFromShapes();
```

这个函数成本较高，所以你应该只在需要时使用它。

你可以轻易地摧毁物体上的一个形状，你可以此来模塑一个可破碎的对象。否则其实你可以忘记那些形状，物体摧毁时那些形状也会摧毁。

```
myBody->DestroyShape(myShape);
```

移除物体上的形状之后，你可能需要再次调用 SetMassFromShapes。

6.4 使用形状

并没有太多需要讲解的东西。你可以得到一个形状的类型和其父物体。另外你也可以测试一个点是否包含于形状之内。细节请查看 b2Shape.h。

7 关节

7.1 关于

关节的作用是把物体约束到世界，或约束到其它物体上。在游戏中的典型例子是木偶，跷跷板和滑轮。关节可以用许多种不同的方法结合起来，创造出有趣的运动。

有些关节提供了限制(limit)，以便你控制运动范围。有些关节还提供了马达(motor)，它可以以指定的速度驱动关节，直到你指定了更大的力或扭矩。

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。

7.2 关节定义

各种关节类型都派生自 b2JointDef。所有关节都连接两个不同的物体，可能其中一个是静态物体。如果你想浪费内存的话，那就创建一个连接两个静态物体的关节 :)

你可以为任何一种关节指定用户数据。你还可以提供一个标记，用于预防相连的物体发生碰撞。实际上，这是默认行为，你可以设置 collideConnected 布尔值来允许相连的物体碰撞。

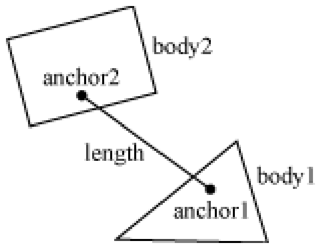
很多关节定义需要你提供一些几何数据。一个关节常常需要一个*锚点*(anchor point)来定义，这是固定于相接物体中的点。在 Box2D 中这些点需要在局部坐标系中指定，这样，即便当前物体的变化违反了关节约束，关节还是可以被指定 —— 在游戏存取进度时这经常会发生。另外，有些关节定义需要默认的物体之间的相对角度。这样才能通过关节限制或固定的相对角来正确地约束旋转。

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。这能使关节行为更加稳固。

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。

7.2.1 距离关节

距离关节是最简单的关节之一，它描述了两个物体上的两个点之间的距离应该是常量。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。随后，你指定两个世界坐标中的*锚点*。第一个锚点连接到物体 1，第二个锚点连接到物体 2。这些点隐含了距离约束的长度。

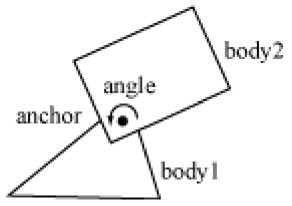


这是一个距离关节定义的例子。在此我们允许了碰撞。

```
b2DistanceJointDef jointDef;
jointDef.Initialize(myBody1, myBody2, worldAnchorOnBody1,
worldAnchorOnBody2);
jointDef.collideConnected = true;
```

7.2.2 旋转关节

一个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一个世界坐标的锚点。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于第一个物体之质心。

```
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，使用那个 Initialize() 创建关节时，旋转关节角为 0，无论两个物体当前的角度怎样。

有时候，你可能需要控制关节角。为此，旋转关节可以模拟关节限制和马达。

关节限制会强制保持关节角度在一个范围内，为此它会应用足够的扭矩。范围内应该包括 0，否则在开始模拟时关节会倾斜。

关节马达允许你指定关节的速度(角度之时间导数)，速度可正可负。马达可以有无穷的力量，但这通常没有必要。你是否听过这句话：

- *注意：*“当一个不可抵抗力遇到一个不可移动物体时会发生什么？”

我可以告诉你这并不有趣。所以你可以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。

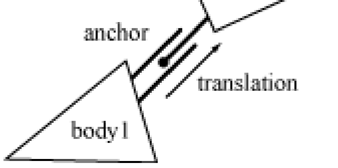
你还可以使用关节马达来模拟关节摩擦。只要把关节速度设置为 0，并设置一个小且有效的最大扭矩即可。这样马达会试图阻止关节旋转，但它会屈服于过大的负载。

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;
```

7.2.3 移动关节

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;
```

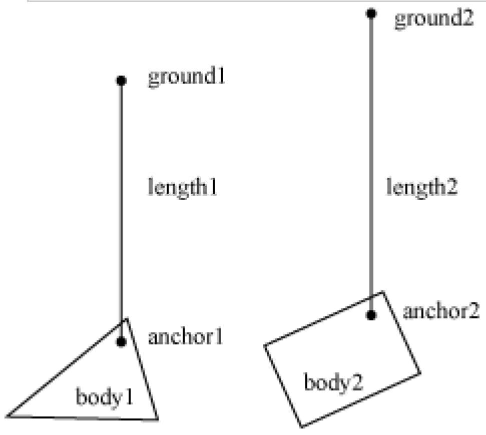
旋转关节隐含着一个从屏幕射出的轴，而移动关节明确地需要一个平行于屏幕的轴。这个轴会固定于两个物体之上，沿着它们的运动方向。

就像旋转关节一样，当使用 Initialize() 创建移动关节时，移动为 0。所以一定要确保移动限制范围内包含了 0。

7.2.4 滑轮关节

滑轮关节用于创建理想的滑轮，它将两个物体接地(ground)并连接到彼此。这样，当一个物体升起时，另一个物体就会下降。滑轮的绳子长度取决于初始时的状态。

```
length1 + length2 == constant
```



你还可以提供一个系数(ratio)来模拟*滑轮组*，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。

当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。

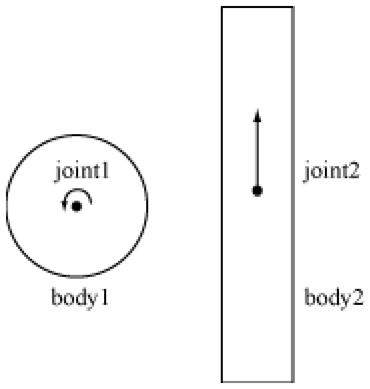
这是一个滑轮定义的例子：

```
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;
```

7.2.5 齿轮关节

如果你想要创建复杂的机械装置，你可能需要齿轮。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，齿轮系数是长度或长度分之一。

```
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，否则你的代码将会由于齿轮关节中的无效关节指针而导致崩溃。另外齿轮关节也应该在任何相关物体被删除之前删除。

7.2.6 鼠标关节

在 testbed 中，鼠标关节用于通过鼠标来操控物体。细节请看 testbed 以及 b2MouseJoint.h。

7.3 关节工厂

关节是通过世界的工厂方法来创建和摧毁的，这引出了一个旧问题：

- *注意：*不要试图在栈上创建物体或关节，也不要使用 new 或 malloc 在堆上创建。物体以及关节必须要通过 b2World 类的方法来创建或摧毁。

这是一个关于旋转关节生命期的例子：

```
b2RevoluteJointDef jointDef;
jointDef.body1 = myBody1;
jointDef.body2 = myBody2;
jointDef.anchorPoint = myBody1->GetCenterPosition();
b2RevoluteJoint* joint = myWorld->CreateJoint(&jointDef);
// ... do stuff ...
myWorld->DestroyJoint(joint);
joint = NULL;
```

摧毁之后将指针清零总是一个好的方式。如果你试图使用它，程序也会以可控的方式崩溃。

关节的生命期并不简单。请留意这个警告：

- *注意：*当物体被摧毁时其上的关节也会摧毁。

并不总是需要这个防范。如果你的游戏引擎会总是在摧毁物体之前摧毁关节，你就无须实现监听类了。更多细节请看 9.2 隐式摧毁。

7.4 使用关节

在许多模拟中，关节被创建之后便不再被访问了。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，或引发其它游戏事件。这些函数可能需要做一些计算，所以不要在不需要的时候调用它们。

```
b2Vec2 GetReactionForce();
float32 GetReactionTorque();
```

7.4.1 使用距离关节

距离关节没有马达以及关节限制，所以它没有额外的运行时方法。

7.4.2 使用旋转关节

你可以访问旋转关节的角度，速度，以及扭矩。

```
float32 GetJointAngle() const;
float32 GetJointSpeed() const;
float32 GetMotorTorque() const;
```

你也可以在每步中更新马达参数。

```
void SetMotorSpeed(float32 speed);
void SetMaxMotorTorque(float32 torque);
```

关节马达有一些有趣的能力。你可以在每个时间步中更新关节速度，这可以使关节像正弦波一样来回移动，或者按其它什么函数运动。

```
// ... Game Loop Begin ...
myJoint->SetMotorSpeed(cosf(0.5f * time));
// ... Game Loop End ...
```

你还可以使用关节马达来追踪某个关节角度。例如：

```
// ... Game Loop Begin ...
float32 angleError = myJoint->GetJointAngle() - angleTarget;
float32 gain = 0.1f;
myJoint->SetMotorSpeed(-gain * angleError);
// ... Game Loop End ...
```

通常来讲你的增益参数不应过大，否则你的关节可能会变得不稳定。

7.4.3 使用移动关节

移动关节的用法类似于旋转关节，这是它的相关成员函数：

```
float32 GetJointTranslation() const;
float32 GetJointSpeed() const;
float32 GetMotorForce() const;
void SetMotorSpeed(float32 speed);
void SetMotorForce(float32 force);
```

7.4.4 使用滑轮关节

滑轮关节提供了当前长度。

```
float32 GetLength1() const;
float32 GetLength2() const;
```

7.4.5 使用齿轮关节

齿轮关节不提供任何多于 b2Joint 中定义的信息。

7.4.6 使用鼠标关节

通过在时间步中更新目标位置，鼠标关节可以控制连接的物体。

8 接触

8.1 关于

接触(contact)是由 Box2D 创建的用于管理形状间碰撞的对象。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。通常这对你并不重要，我只是想或许你愿意了解一些。

这里是 Box2D 中的一些与碰撞有关的术语，但你也可能会在其它物理引擎中发现类似的术语。

触点(contact point)

两个形状相互接触的点。实际上当物体的表面相接触时可能会有一定接触区域，在 Box2D 则近似地以少数点来接触。

接触向量(contact normal)

从 shape1 指向 shape2 的单位向量。

接触分隔(contact separation)

分隔相反于穿透，当形状相重叠时，分隔为负。可能以后的 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会检查符号。

法向力(normal force)

Box2D 使用了一个迭代接触求解器，并会以触点保存结果。你可以安全地使用法向力来判断碰撞强度。例如，你可以使用这个力来引发破碎，或者播放碰撞的声音。

切向力(tangent force)

它是接触求解器关于摩擦力的估计量。

接触标识(contact ids)

Box2D 会试图利用一个时间步中的触点压力(contact force)结果来推测下一个时间步中的情况。接触标识用于匹配跨越时间步的触点，它包含了几何特征索引以便区分触点。

当两个形状的 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，有时尽管碰撞已筛选了 Box2D 还是须要创建一个接触，这种情况下它会使用 b2NullContact 来防止碰撞的发生。当 AABB 不再重叠之后接触会被摧毁。

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。Box2D 选择了后面这个方法。

8.2 接触监听器

通过实现 b2ContactListener 你就可以接受接触数据。当一个触点被创建时，当它持续超过一个时间步时，以及当它被摧毁时，这个监听器(listener)就会发出报告。请留意两个形状之间可能会有多个触点。

```
class MyContactListener : public b2ContactListener
{
public:
 void Add(const b2ContactPoint* point)
 {
 // handle add point
 }

 void Persist(const b2ContactPoint* point)
 {
 // handle persist point
 }

 void Remove(const b2ContactPoint* point)
 {
 // handle remove point
 }

 void Result(const b2ContactResult* point)
 {
 // handle results
 }
};
```

- *注意：*不要保存 b2ContactListener 中触点的引用，取而代之，用深拷贝将触点数据保存到你自己的缓冲区中。下面的例子演示了一种方法。

连续性的物理模拟使用了子步，所以一个触点可能会创建和摧毁于同一个时间步中。通常这不是问题，但你的代码应该温雅地处理它。

当触点创建，持续或删除时，会有即刻的报告。这出现于求解器调用之前，所以 b2ContactPoint 并不包含已计算的冲量。然而，触点处的相对速度提供了，这样你可以估计出接触冲量。如果你实现了结果监听函数，那么在求解器调用之后你就会收到 b2ContactResult 对象，包含了可靠的触点信息。这些保存结果的结构中包含了子步的冲量。重申一次，由于连续性的物理模拟，在一个 b2World::Step 中你可能会收到单个触点的多个结果。 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。

```
// We are going to destroy some bodies according to contact
// points. We must buffer the bodies that should be destroyed
// because they may belong to multiple contact points.
const int32 k_maxNuke = 6;
b2Body* nuke[k_maxNuke];
int32 nukeCount = 0;

// Traverse the contact buffer. Destroy bodies that
// are touching heavier bodies.
for (int32 i = 0; i < m_pointCount; ++i)
{
 ContactPoint* point = m_points + i;

 b2Body* 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.
std::sort(nuke, nuke + nukeCount);

// Destroy the bodies, skipping duplicates.
int32 i = 0;
while (i < nukeCount)
{
 b2Body* b = nuke[i++];
 while (i < nukeCount && nuke[i] == b)
 {
 ++i;
 }

 m_world->DestroyBody(b);
}
```

8.3 接触筛选

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某些角色才能通过的门。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。这个类需要一个 ShouldCollide 函数，用于接收两个 b2Shape 的指针，如果应该碰撞那么就返回 true。

默认的 ShouldCollide 实现使用了 6 形状 中的 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;
}
```

9 杂项

9.1 世界边界

你可以实现一个 b2BoundaryListener，这样当有物体超出世界的 AABB 时 b2World 就能通知你。当你得到回调时，你不应该试图删除物体；取而代之的是，你可以为角色做个删除或错误处理标记，在物理时间步之后再进行这个事件的处理。

```
class MyBoundaryListener : public b2BoundaryListener
{
 void Violation(b2Body* body)
 {
 MyActor* myActor = (MyActor*)body->GetUserData();
 myActor->MarkForErrorHandling();
 }
};
```

随后你可以在世界对象中注册你的边界监听器实例，这应该安排在世界初始化过程中。

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

9.2 隐式摧毁

Box2D 并不使用引用计数。所以当你摧毁一个物体后，它的确就不存在了。以指针访问一个已摧毁物体的行为是未定义的，换句话说，你的程序可能会崩溃。有些时候，调试模式的内存管理器可能会帮助找到这些问题。

如果你摧毁一个 Box2D 实体，你应该保证所有到它的引用都删除了。如果你只有实体的单个引用的话，那就简单了。但如果你有很多个引用，你可能要考虑实现一个*处理*类来封装原始指针。

通常使用 Box2D 时你需要创建并摧毁许多物体，形状还有关节。管理这些实体有些自动化，如果你摧毁一个物体，所有它的形状，关节，以及接触都会摧毁，这称为*隐式摧毁*。任何连接于这些关节或接触之一的物体将被唤醒，通常这是便利的。然而，你应该意识到了一个关键问题：

- *注意：*当一个物体摧毁时，所有它的形状和关节都会自动摧毁。你应该将任何指向这些形状或关节的指针置零，否则之后如果你试图访问它们，你的程序会崩溃。

Box2D 提供了一个名为 b2WorldListener 的监听器类，你可以实现它并提供给世界对象，随后当关节将被隐式摧毁时世界对象就会提醒你。

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，这可以帮助你预防访问无效指针。

```
class MyDestructionListener : public b2DestructionListener
{
 void SayGoodbye(b2Joint* joint)
 {
 // remove all references to joint.
 }
};
```

随后你可以注册它，这应该在世界初始化过程中。

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

10 设置

10.1 关于

Box2D 为用户提供了 b2Settings.h 和 b2Settings.cpp 两个定制源文件。

Box2D 使用浮点数工作，所以使用了一些公差，以便能良好的运行。

10.2 公差

许多设置依赖于 MKS 单位，在 3.3 单位 中有更多解释。个别公差解释请看 doxygen 文档。

10.3 内存分配

除了下面的情况以外，所有 Box2D 中的内存分配都是通过 b2Alloc 和 b2Free 完成的。

- b2World 可以在栈上建造，或任意你喜欢的地方。
- 其它非工厂方法创建的 Box2D 类，包括回调类和触点缓冲。

可以自由修改 b2Settings.cpp 来改变分配动作。

11 调试绘图

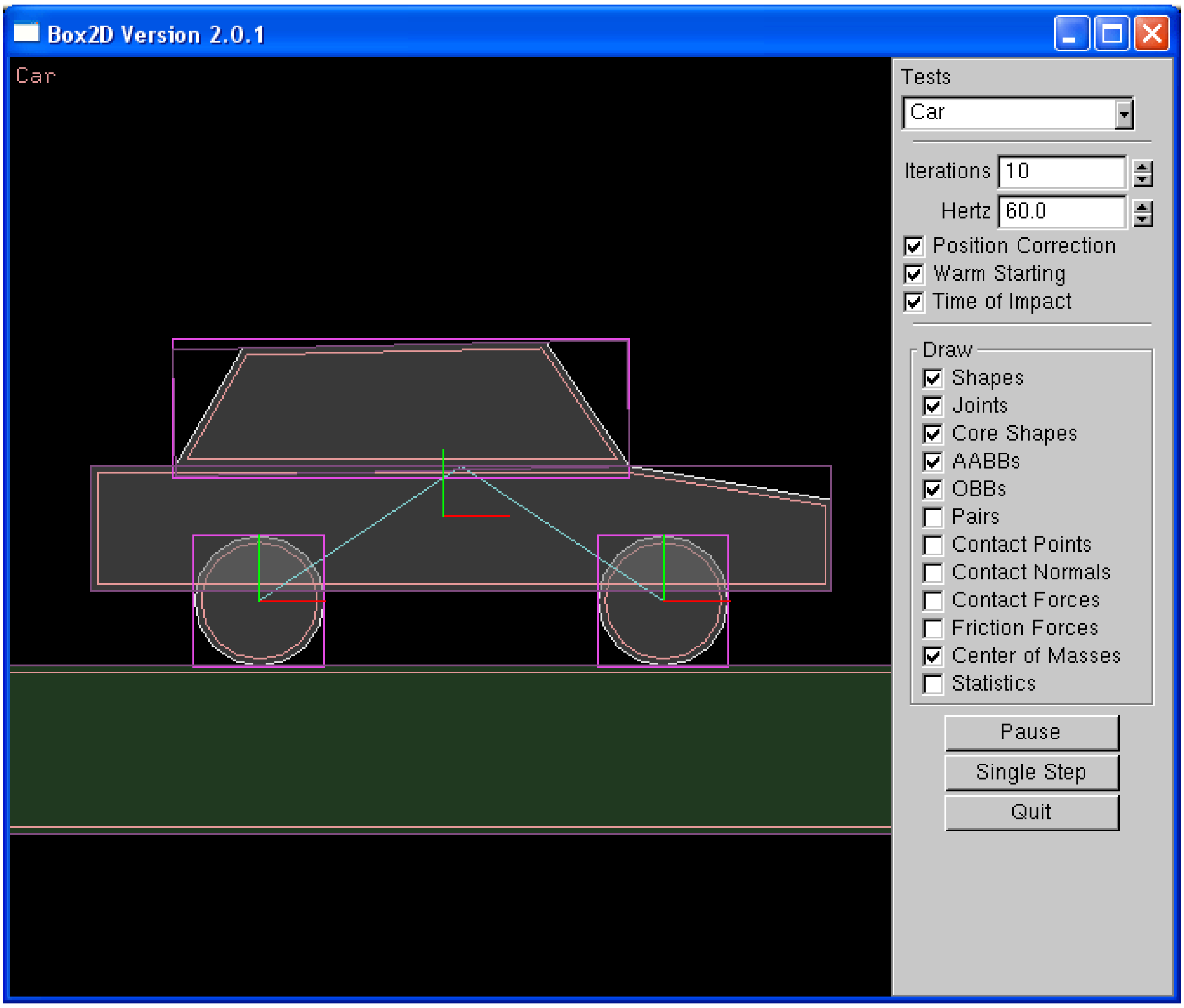
实现 b2DebugDraw 可得到物理世界的细部图，这里是可用的实体：

- 形状轮廓

- 关节连通性
- 核心形状(为连续碰撞)
- broad-phase AABB，包括世界 AABB
- 多边形 OBB
- broad-phase pair(潜在接触)
- 质心

这是绘制这些物理实体的首选方法，优于直接访问数据。理由是许多必要信息都是内在的。

testbed 使用了调试绘图设施以及接触监听器来绘制物理实体，所以它是实现调试绘图的主要例子。



Tests: Car

Iterations: 10

Hertz: 60.0

- ✓ Position Correction
- ✓ Warm Starting
- ✓ Time of Impact

Draw

- ✓ Shapes
- ✓ Joints
- ✓ Core Shapes
- ✓ AABBs
- ✓ OBBs
- ☐ Pairs
- ☐ Contact Points
- ☐ Contact Normals
- ☐ Contact Forces
- ☐ Friction Forces
- ✓ Center of Masses
- ☐ Statistics

简体中文翻译信息

- 原文：Box2D v2.0.2 User Manual
- 中文译者：Aman JIANG(江超宇)
- 主页：lesslab.com 电子邮件：amanjiang@gmail.com

Chapter 1 Introduction

1.1 About

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's point of view a physics engine is just a system for procedural animation.

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.

1.2 Prerequisites

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 the many tutorials provided by Chris Hecker and David Baraff (Google these names). You do not need to understand their tutorials in great detail, but they do a good job of laying out the basic concepts that will help you use Box2D.

Wikipedia is also an excellent source of physics and mathematics knowledge. In some ways it is more useful than Google, because it has carefully crafted content.

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.

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.

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.

1.3 About this Manual

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.

1.4 Feedback and Reporting Bugs

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

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.

File issues 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.

1.5 Core Concepts

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

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

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

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

constraint
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.

contact constraint
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.

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.

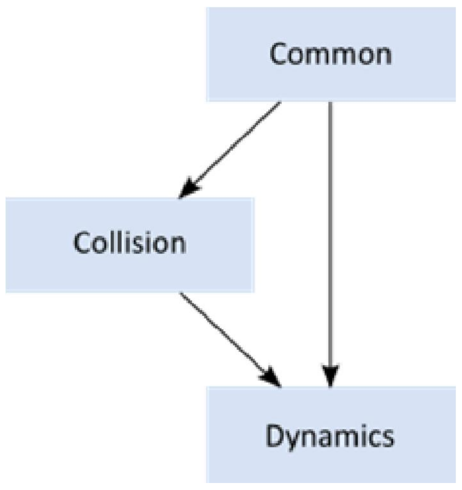
joint limit
A joint limit restricts the range of motion of a joint. For example, the human elbow only allows a certain range of angles.

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.

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.

1.6 Modules

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.



1.7 Units

Box2D works with floating point numbers, so some 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 without too much trouble.

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. Imagine trying to simulate the movement of a high-rise building with an engine that is tuned to simulate ragdolls and barrels. It isn't pretty.

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.

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.

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).

1.8 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.

There are creation functions:

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

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

And there are corresponding destruction functions:

```
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.

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

```
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.

```
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.

1.9 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 data structures in your game engine.

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.

```
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:

- 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.

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.

Chapter 2 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.

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

2.1 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. The decision is up to you.

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.

```
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.

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

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

2.2 Creating a Ground Box

Bodies are built using the following steps:

1. Define a body with a 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.

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.

```
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. The ground body is created as a static body. Static bodies don't collide with other static bodies and are immovable. Box2D determines that a body is static when it has zero mass. Bodies have zero mass by default; therefore they are static by default.

```
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.

```
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.

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 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.

```
groundBody->CreateFixture(&groundBox);
```

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

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. This need for static bodies is done to make the Box2D code more uniform internally, reducing the number of potential bugs.

You might notice a pattern here. Most Box2D types are prefixed with b2. This is done to reduce the chance for naming conflicts with your code.

2.3 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.

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.

```
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:

```
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.

```
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.

```
body->CreateFixture(&fixtureDef);
```

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

2.4 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.

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.

```
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.

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.

The suggested iteration count for Box2D is 10 for both velocity and 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.

```
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 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.

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. After stepping, you should call b2World::ClearForces to clear any forces you applied to the bodies.

The Hello World program was designed to be dead simple, so it has no graphical output. Rather than being utterly boring by producing no 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.

```
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);

}
```

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

```
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 Cleanup

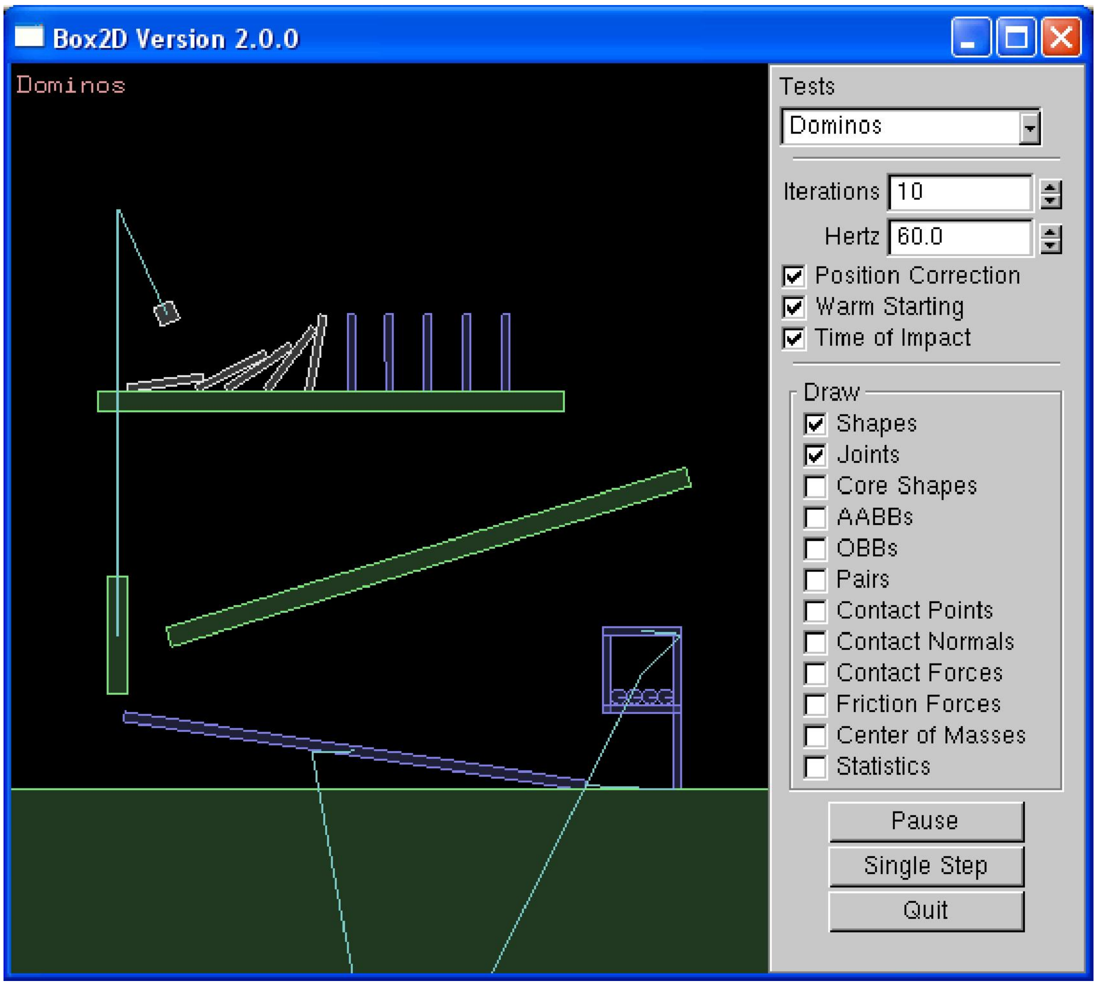
When a world leaves scope or is deleted by calling delete on a pointer, all the memory reserved for bodies, fixtures, and joints is freed. This is done to make your life easier. However, you will need to nullify any body, fixture, or joint pointers you have because they will become invalid.

2.6 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:

- 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.


Box2D Version 2.0.0

Dominos

Tests: Dominos

Iterations: 10

Hertz: 60.0

- [✓] Position Correction
- [✓] Warm Starting
- [✓] Time of Impact

Draw

- [✓] Shapes
- [✓] Joints
- [ ] Core Shapes
- [ ] AABBs
- [ ] OBBs
- [ ] Pairs
- [ ] Contact Points
- [ ] Contact Normals
- [ ] Contact Forces
- [ ] Friction Forces
- [ ] Center of Masses
- [ ] Statistics

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.

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.

Chapter 3 Common

3.1 About

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

3.2 Settings

The header b2Settings.h contains:

- Types such as int32 and float32
- Constants
- Allocation wrappers
- The version number
- Friction and restitution mixing functions

Types

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

Constants

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

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.

Allocation wrappers

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

Version

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

Friction and Restitution Mixing

These are placed in the settings file in case you need to customize them for your application.

3.3 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.

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'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.

Since Box2D uses a SOA, you should never new or malloc a body, fixture, or joint. You do have to allocate a b2World. 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, ever, call delete or free on a body, fixture, or joint.

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.

3.4 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.

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

Chapter 4 Collision Module

4.1 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.

4.2 Shapes

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

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

- 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.

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

4.3 Circle Shapes

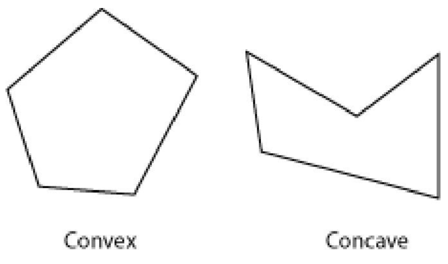
Circle shapes have a position and radius.

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

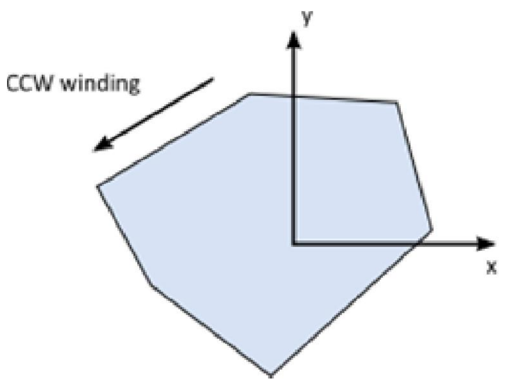
```
b2CircleShape circle;

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

circle.m_radius = 0.5f;
```

4.4 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. However, you can create line segments using 2 point polygons.



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.



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

You can create a polygon shape by passing in a vertex array. The maximal size of the array is controlled by b2_maxPolygonVertices which has a default value of 8. This is sufficient to describe most convex polygons.

```
// 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 and edges (line segments).

```
void SetAsBox(float32 hx, float32 hy);

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

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

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 Shape Point Test

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

```
b2Transfrom transform;

transform.SetIdentity();

b2Vec2 point(5.0f, 2.0f);

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

4.6 Shape Ray Cast

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.

```
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;
b2RayCastOutput output;
bool hit = shape->RayCast(&output, input, transform);
if (hit)
{
  b2Vec2 hitPoint = input.p1 + output.fraction * (input.p2 – input.p1);
  …
}
```

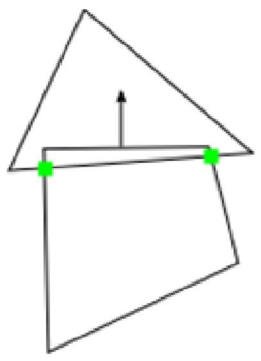
4.7 Bilateral Functions

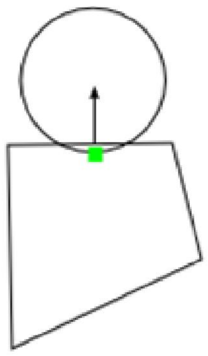
The Collision module contains bilateral functions that take a pair of shapes and compute some results. These include:

- Contact manifolds
- Distance
- Time of impact

4.8 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.


two points, one normal


two points, one normal

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

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.

The b2WorldManifold structure can be used to generate the world coordinates of the contact normal and points. You need to provide a b2Manifold and the shape transforms and radii.

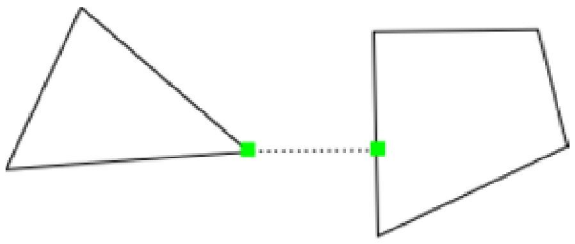
```
b2WorldManifold worldManifold;

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.

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

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

if (state1[0] == b2_removeState)

{

  // process event

}
```

4.9 Distance

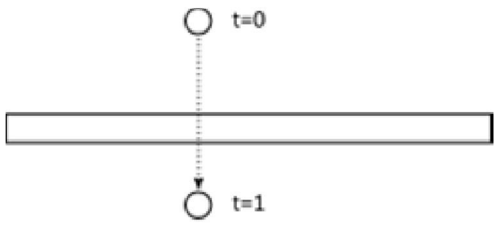
The b2Distance function can be used to compute the distance between two shapes. The distance function needs both shapes to be converted into a b2DistanceProxy. There is also some caching used to warm start the distance function for repeated calls. You can see the details in b2Distance.h.



distance and closest points

4.10 Time of Impact

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

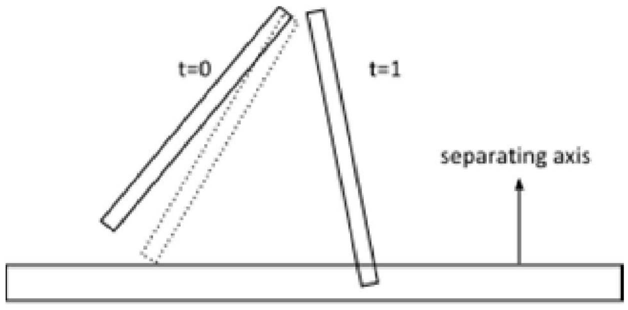


tunneling

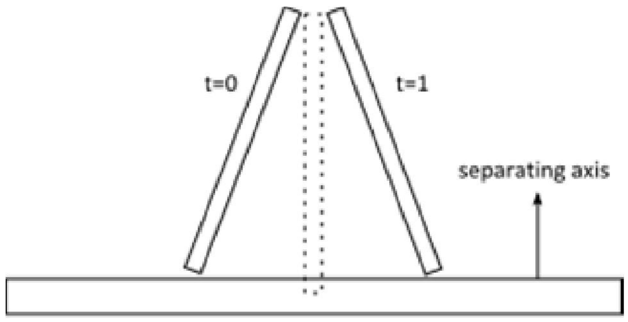
The b2TimeOfImpact is used to determine the time when two moving shapes collide. This is called the *time of impact* (TOI). The main purpose of b2TimeOfImpact is for tunnel prevention. In particular, it is designed to prevent moving objects from tunneling outside of static level geometry.

This function 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.

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.



captured collision



missed collision

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.

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

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

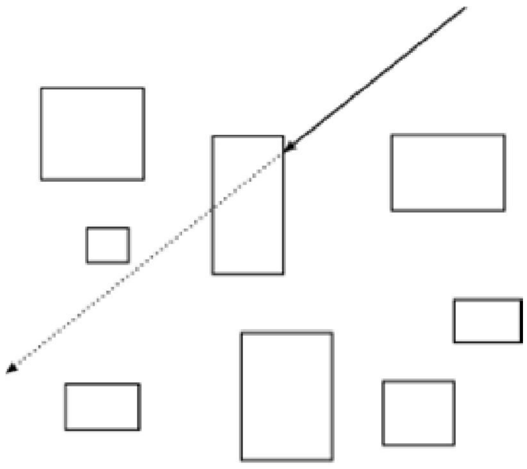
4.11 Dynamic Tree

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.

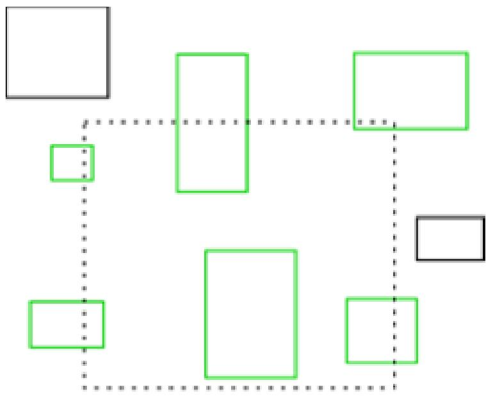
The dynamic tree is a hierarchical AABB tree. Each internal node in the tree can has two children. A leaf node is a single user AABB.

The tree 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.

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.



ray cast



region query

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.

4.12 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.

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

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.

Chapter 5 Dynamics Module

5.1 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.

The Dynamics module contains:

- 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.

The dynamics module is covered in the following chapters.

Chapter 6 Fixtures

6.1 About

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

- a single shape
- density, friction, and restitution
- collision filtering flags
- back pointer to parent body
- user data
- sensor flag

These are described in the following sections.

6.2 Fixture Creation

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

```
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.

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.

```
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.

The mass of a body is automatically adjusted when you

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:

```
float32 friction;

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

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

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.

```
float32 restitution;

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

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

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.

Filtering

Collision filtering is a system for preventing 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.

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:

```
playerFixtureDef.filter.categoryBits = 0x0002;

monsterFixtureDef.filter.categoryBits = 0x0004;

playerFixtureDef.filter.maskBits = 0x0004;

monsterFixtureDef.filter.maskBits = 0x0002;
```

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.

```
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.

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

- A fixture on a static body never collides with a fixture on another static 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.

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).

6.3 Sensors

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.

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.

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

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

Chapter 7 Bodies

7.1 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:

b2_staticBody
A static body has 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.

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.

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 kilogram.

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.

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.

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.

7.2 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.

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.

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

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.

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

Setting the body type is mandatory.

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.

> **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.

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.

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.

```
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.

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.

```
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).

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.

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.

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

```
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:

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

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

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.

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.

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.

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.

```
bodyDef.bullet = true;
```

The bullet flag only affects dynamic bodies.

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

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.

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

```
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.

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.

```
b2BodyDef bodyDef;

bodyDef.userData = &myActor;
```

7.3 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.

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

```
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.

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.

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 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.

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.

7.4 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.

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.

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.

```
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:

```
void ResetMassData();
```

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

```
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:

```
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.

```
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.

```
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.

Chapter 8 Joints

8.1 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.

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.

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.

8.2 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.

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.

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.

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.

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

8.3 Joint Factory

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

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.

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

```
b2RevoluteJointDef jointDef;

jointDef.body1 = myBody1;

jointDef.body2 = myBody2;

jointDef.anchorPoint = myBody1->GetCenterPosition();

b2RevoluteJoint* joint = 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.

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

Caution

Joints are destroyed when an attached body is destroyed.

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

8.4 Using Joints

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

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

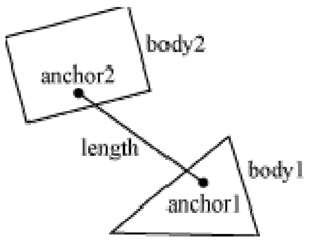
```
b2Body* GetBody1();

b2Body* GetBody2();

b2Vec2 GetAnchor1();

b2Vec2 GetAnchor2();

void* GetUserData();
```

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

```
b2Vec2 GetReactionForce();

float32 GetReactionTorque();
```

8.5 Distance Joint

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



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

```
b2DistanceJointDef jointDef;

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

jointDef.collideConnected = true;
```

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

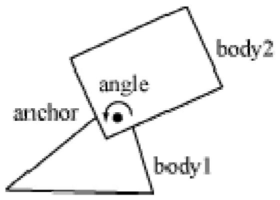
Softness is achieved by tuning two constants in the definition: frequency and damping ratio. Think of the frequency as the frequency of a harmonic oscillator (like a guitar string). The frequency is specified in Hertz. Typically the frequency should be less than a half the frequency of the time step.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

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

```
jointDef.frequencyHz = 4.0f;

jointDef.dampingRatio = 0.5f;
```

8.6 Revolute Joint

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



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

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

```
b2RevoluteJointDef jointDef;

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

The revolute joint angle is positive when body2 rotates CCW about the angle point. Like all angles in Box2D, the revolute angle is measured in radians. By convention the revolute joint angle is zero when the joint is created using Initialize(), regardless of the current rotation of the two bodies.

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

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

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

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

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

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

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

```
b2RevoluteJointDef jointDef;

jointDef.Initialize(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;
```

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

```
float32 GetJointAngle() const;

float32 GetJointSpeed() const;

float32 GetMotorTorque() const;
```

You also update the motor parameters each step.

```
void SetMotorSpeed(float32 speed);

void SetMaxMotorTorque(float32 torque);
```

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

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

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

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

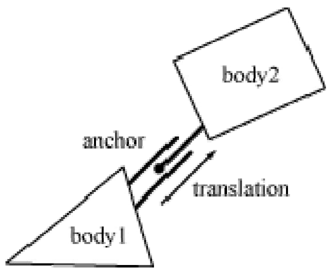
You can also use joint motors to track a desired joint angle. For example:

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

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

float32 gain = 0.1f;

myJoint->SetMotorSpeed(-gain * angleError);

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

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

8.7 Prismatic Joint

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



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

```
b2PrismaticJointDef jointDef;

b2Vec2 worldAxis(1.0f, 0.0f);

jointDef.Initialize(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;
```

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

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

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

```
float32 GetJointTranslation() const;

float32 GetJointSpeed() const;

float32 GetMotorForce() const;

void SetMotorSpeed(float32 speed);

void SetMotorForce(float32 force);
```

8.8 Pulley Joint

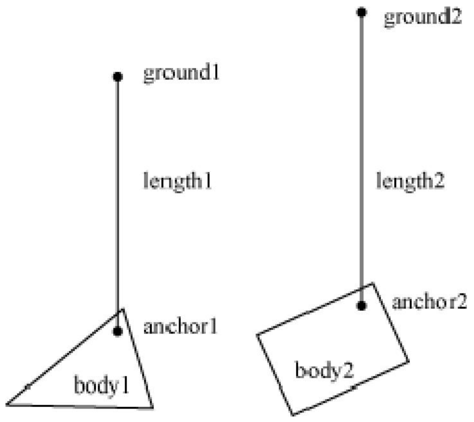
A pulley is used to create an idealized pulley. The pulley connects two bodies to ground and to each other. As one body goes up, the other goes down. The total length of the pulley rope is conserved according to the initial configuration.

```
length1 + length2 == constant
```

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

```
length1 + ratio * length2 == constant
```

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



Pulleys can be troublesome when one side is fully extended. The rope on the other side will have zero length. At this point the constraint equations become singular (bad). Therefore the pulley joint constrains the maximum length that either side can attain. Also, you may want to control the maximum lengths for game play reasons. So the maximum lengths improve stability and give you more control.

Here is an example pulley definition:

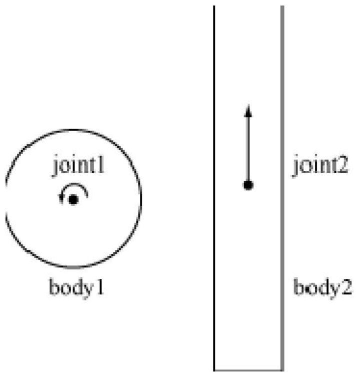
```
b2Vec2 anchor1 = myBody1->GetWorldCenter();

b2Vec2 anchor2 = myBody2->GetWorldCenter();

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

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

float32 ratio = 1.0f;

b2PulleyJointDef jointDef;

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

jointDef.maxLength1 = 18.0f;

jointDef.maxLength2 = 20.0f;
```

Pulley joints provide the current lengths.

```
float32 GetLength1() const;

float32 GetLength2() const;
```

8.9 Gear Joint

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



The gear joint requires that you have two bodies connected to ground by a revolute or prismatic joint. You can use any combination of those joint types. Also, Box2D requires that the revolute and prismatic joints were created with the ground as body1.

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

```
coordinate1 + ratio * coordinate2 == constant
```

Here is an example gear joint:

```
b2GearJointDef jointDef;

jointDef.body1 = myBody1;

jointDef.body2 = myBody2;

jointDef.joint1 = myRevoluteJoint;

jointDef.joint2 = myPrismaticJoint;

jointDef.ratio = 2.0f * b2_pi / myLength;
```

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

Caution

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

8.10 Mouse Joint

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

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

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

8.11 Line Joint

The line joint is like the prismatic joint with the rotation restriction removed. This was requested by a user to model a vehicle wheel with a suspension. See b2LineJoint.h for details.

8.12 Weld Joint

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

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

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

Chapter 9 Contacts

9.1 About

Contacts are objects created by Box2D to manage collision between fixtures. There are different kinds of contacts, derived from b2Contact, for managing contact between different kinds of fixtures. For example there is a contact class for managing polygon-polygon collision and another contact class for managing circle-circle collision.

Here is some terminology associated with contacts.

contact point

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

contact normal

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

contact separation

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

contact manifold

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

normal impulse

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

tangent impulse

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

contact ids

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

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

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

9.2 Contact Class

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

You can access the raw contact manifold:

```
b2Manifold* GetManifold();

const b2Manifold* GetManifold() const;
```

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

There is a helper function to get the b2WorldManifold:

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

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

Sensors do not create manifolds, so for them use:

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

This function also works for non-sensors.

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

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

b2Body* bodyA = fixtureA->GetBody();

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

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

9.3 Accessing Contacts

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

You can iterate over all contacts in the world:

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

{

  // process c

}
```

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

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

{

  b2Contact* c = ce->contact;

  // process c

}
```

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

Caution

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

9.4 Contact Listener

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

```
class MyContactListener : public b2ContactListener

{

public:

  void BeginContact(b2Contact* contact)

  { // handle begin event }


  void EndContact(b2Contact* contact)

  { // handle end event }


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

  { // handle pre-solve event }


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

  { // handle post-solve event }

};
```

Caution

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

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

Begin Contact Event

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

End Contact Event

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

Pre-Solve Event

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

```
void PreSolve(b2Contact* contact, const b2Manifold* oldManifold)
{
  b2WorldManifold worldManifold;
  contact->GetWorldManifold(&worldManifold);
  if (worldManifold.normal.y < -0.5f)
  {
    contact->SetEnabled(false);
  }
}
```

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

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

{

  b2WorldManifold worldManifold;

  contact->GetWorldManifold(&worldManifold);

  b2PointState state1[2], state2[2];

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

  if (state2[0] == b2_addState)

  {

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

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

    b2Vec2 point = worldManifold.points[0];

    b2Vec2 vA = bodyA->GetLinearVelocityFromWorldPoint(point);

    b2Vec2 vB = bodyB->GetLinearVelocityFromWorldPoint(point);

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

    if (approachVelocity > 1.0f)

    {

      MyPlayCollisionSound();

    }

  }

}
```

Post-Solve Event

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

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

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

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

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

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

// because they may belong to multiple contact points.

const int32 k_maxNuke = 6;

b2Body* nuke[k_maxNuke];

int32 nukeCount = 0;



// Traverse the contact buffer. Destroy bodies that

// are touching heavier bodies.

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

{

    ContactPoint* point = m_points + i;



    b2Body* 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.
std::sort(nuke, nuke + nukeCount);

// Destroy the bodies, skipping duplicates.
int32 i = 0;
while (i < nukeCount)
{
    b2Body* b = nuke[i++];
    while (i < nukeCount && nuke[i] == b)
    {
        ++i;
    }

    m_world->DestroyBody(b);
}
```

9.5 Contact Filtering

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

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

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

```
bool b2ContactFilter::ShouldCollide(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;
}
```

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

```
MyContactFilter filter;

world->SetContactFilter(&filter);

// filter remains in scope …
```

Chapter 10 World Class

About

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

Creating and Destroying a World

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

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

... do stuff ...

delete myWorld;
```

Using a World

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

Simulation

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

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

int32 velocityIterations = 10;

int32 positionIterations = 8;

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

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

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

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

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

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

Exploring the World

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

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

{

    b->WakeUp();

}
```

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

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

{

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

    if (myActor->IsDead())

    {

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

    }

}
```

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

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

while (node)

{

    b2Body* b = node;

    node = node->GetNext();



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

    if (myActor->IsDead())

    {

        myWorld->DestroyBody(b);

    }

}
```

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

```
b2Body* node = myWorld->GetBodyList();
while (node)
{
    b2Body* b = node;
    node = node->GetNext();

    GameActor* myActor = (GameActor*)b->GetUserData();
    if (myActor->IsDead())
    {
        bool otherBodiesDestroyed = GameCrazyBodyDestroyer(b);
        if (otherBodiesDestroyed)
        {
            node = myWorld->GetBodyList();
        }
    }
}
```

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

AABB Queries

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

```
class MyQueryCallback : public b2QueryCallback
{
public:
    bool ReportFixture(b2Fixture* fixture)
    {
        b2Body* body = fixture->GetBody();
        body->WakeUp();

        // Return true to continue the query.
        return true;
    }
};

...

MyQueryCallback callback;
b2AABB aabb;
aabb.lowerBound.Set(-1.0f, -1.0f);
aabb.upperBound.Set(1.0f, 1.0f);
myWorld->Query(&callback, aabb);
```

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

Ray Casts

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

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

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

Here is an example:

```
// This class captures the closest hit shape.
class MyRayCastCallback : public b2RayCastCallback
{
public:
    MyRayCastCallback()
    {
        m_fixture = NULL;
    }

    float32 ReportFixture(b2Fixture* fixture, const b2Vec2& point,
                                          const b2Vec2& normal, float32 fraction)
    {
        m_fixture = fixture;
        m_point = point;
        m_normal = normal;
        m_fraction = fraction;
        return fraction;
    }

    b2Fixture* m_fixture;
    b2Vec2 m_point;
    b2Vec2 m_normal;
    float32 m_fraction;
};

MyRayCastCallback callback;
b2Vec2 point1(-1.0f, 0.0f);
```

```
b2Vec2 point2(3.0f, 1.0f);

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

Caution

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

```
void SetLinearVelocity(const b2Vec2& v);

b2Vec2 GetLinearVelocity() const;

void SetAngularVelocity(float32 omega);

float32 GetAngularVelocity() const;
```

Forces and Impulses

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

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

void ApplyTorque(float32 torque);

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

void ApplyAngularImpulse(float32 impulse);
```

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

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

{

    myBody->ApplyForce(myForce, myPoint);

}
```

Coordinate Transformations

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

```
b2Vec2 GetWorldPoint(const b2Vec2& localPoint);

b2Vec2 GetWorldVector(const b2Vec2& localVector);

b2Vec2 GetLocalPoint(const b2Vec2& worldPoint);

b2Vec2 GetLocalVector(const b2Vec2& worldVector);
```

Lists

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

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

{

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

    ... do something with data ...

}
```

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

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

Chapter 11 Loose Ends

11.1 Implicit Destruction

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

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

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

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

Caution

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

To help you nullify your joint pointers, Box2D provides a listener class named b2WorldListener 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.

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

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

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

```
class MyDestructionListener : public b2DestructionListener

{

    void SayGoodbye(b2Joint* joint)

    {

        // remove all references to joint.

    }

};
```

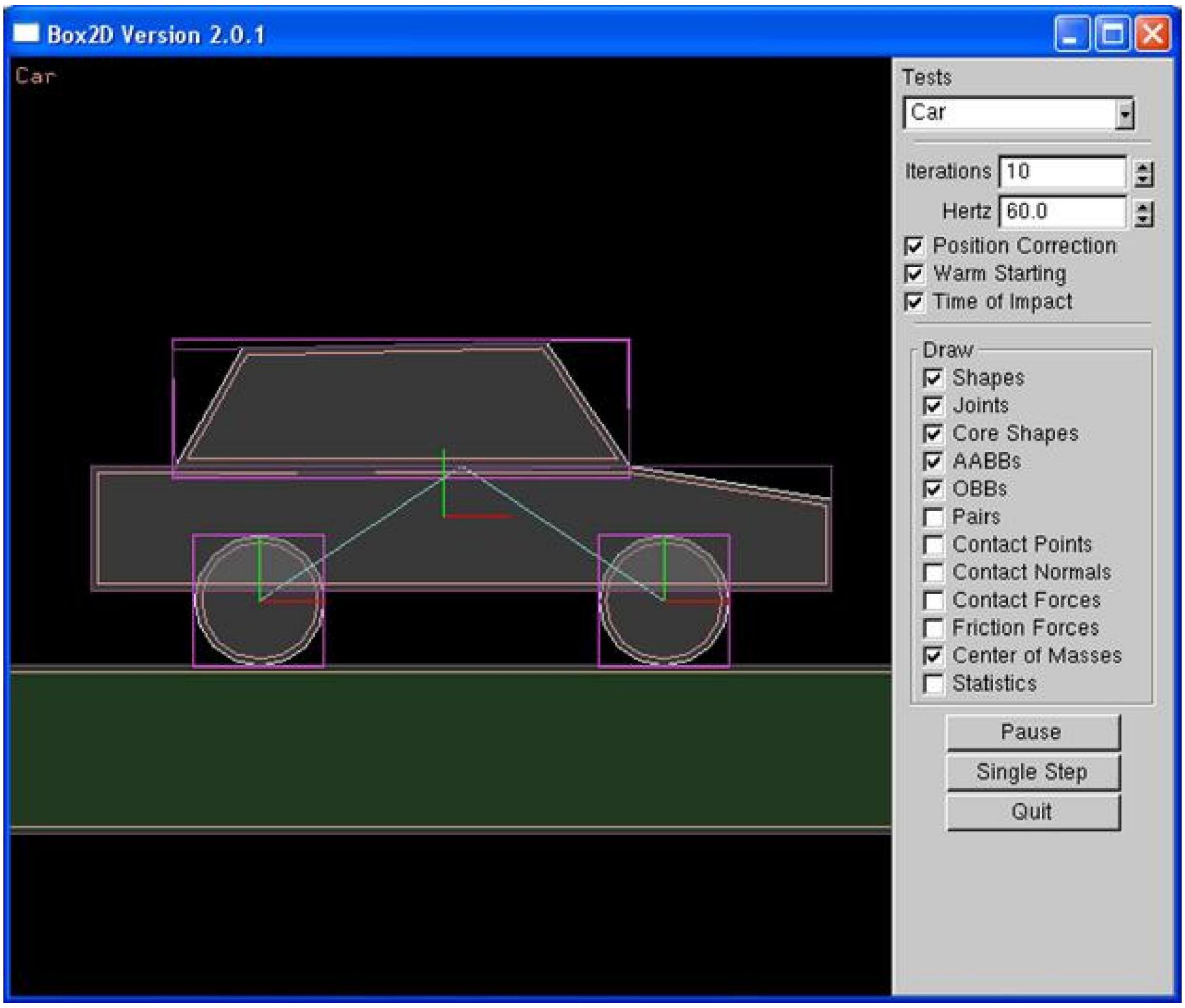
You can then register an instance of your destruction listener with your world object. You should do this during world initialization.

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

Chapter 12 Debug Drawing

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

- shape outlines
- joint connectivity
- core shapes (for continuous collision)
- broad-phase axis-aligned bounding boxes (AABBs), including the world AABB
- polygon oriented bounding boxes (OBBs)
- broad-phase pairs (potential contacts)
- center of mass



Tests: Car

Iterations 10

Hertz 60.0

- ✓ Position Correction
- ✓ Warm Starting
- ✓ Time of Impact

Draw

- ✓ Shapes
- ✓ Joints
- ✓ Core Shapes
- ✓ AABBs
- ✓ OBBs
- ☐ Pairs
- ☐ Contact Points
- ☐ Contact Normals
- ☐ Contact Forces
- ☐ Friction Forces
- ✓ Center of Masses
- ☐ Statistics

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

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

Chapter 13 Limitations

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

Here are the current limitations:

1. Stacking heavy bodies on top of much lighter bodies is not stable. Stability degrades as the mass ratio passes 10:1.
2. 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. This problem will be addressed in the future.
3. 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.
4. There is typically around 0.5cm of slop in shape versus shape collision.
5. Continuous collision is processed sequentially. In a time of impact event, the body is moved back and held there for the rest of the time step. This may make fast moving objects appear to move in a non-smooth manner.

Chapter 14 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