Box2D v2.3.0 User Manual

Copyright © 2007-2013 Erin Catto

[Chapter 1 Introduction 2](#_Toc258082967)

[Chapter 2 Hello Box2D 7](#_Toc258082968)

[Chapter 3 Common 13](#_Toc258082969)

[Chapter 4 Collision Module 15](#_Toc258082970)

[Chapter 5 Dynamics Module 22](#_Toc258082971)

[Chapter 6 Fixtures 23](#_Toc258082972)

[Chapter 7 Bodies 26](#_Toc258082973)

[Chapter 8 Joints 33](#_Toc258082974)

[Chapter 9 Contacts 42](#_Toc258082975)

[Chapter 10 World Class 49](#_Toc258082976)

[Chapter 11 Loose Ends 55](#_Toc258082977)

[Chapter 12 Debug Drawing 57](#_Toc258082978)

[Chapter 13 Limitations 59](#_Toc258082979)

[Chapter 14 References 60](#_Toc258082980)

# Introduction

## Box2D에 대해

Box2D는 게임을 위한 2D rigid body simulation 라이브러리이다. 프로그래머들은 그들의 게임에서 오브젝트들을 현실적인 방법으로 움직이게 하고, 그 게임 월드가 좀 더 상호작용하게 만들기 위해 이것을 사용할 수 있다. 게임 엔진의 관점에서, 물리 엔진은 절차적 애니메이션을 위한 한 시스템이다.

Box2D는 포팅 가능한 C++로 쓰여졌다. 엔진에서 정의된 대부분의 types들은 b2라는 접두어로 시작한다. 바라건데, 이렇게 하는 것이 너의 게임엔진과 충돌하는 이름을 피하기에 충분할 것이다.

## 전제조건

이 매뉴얼에서, 나는 너가 질량, force, torque, 그리고 impulse와 같은 기본 물리 개념에 친숙하다고 가정할 것이다. 만약 그렇지 않다면, 구글 검색과 위키피디아를 찾아보아라.

Box2D는 Game Developer Conference(GDC)에서 physics tutorial의 일부로 만들어졌었다. 너는 이러한 튜토리얼을 box2d.org의 다운로드 섹션에서 얻을 수 있다.

Box2D는 C++로 쓰여있기 때문에, C++ 프로그래밍 경험이 있는 것이 기대된다. Box2D는 너의 첫 번째 C++ 프로그래밍 프로젝트가 되어서는 안된다. 너는 컴파일, linking, 그리고 디버깅에 어려움이 없어야 한다.

**경고**

Box2D는 너의 첫 번째 C++ 프로젝트가 되어서는 안된다. Box2D로 작업하기 전에 C++ 프로그래밍, 컴파일, linking, 그리고 디버깅을 배워라. 이것을 위해 인터넷에 많은 자료들이 있다.

## 이 매뉴얼에 대해

이 매뉴얼은 대부분의 Box2D API를 다룬다. 그러나, 모든 면들이 다뤄지는 것은 아니다. 좀 더 배우기 위해서 Box2D와 함께 포함된 testbed를 보는 것이 격려된다. 또한, Box2D 코드는 Doxygen을 위해 포맷화된 주석들을 가지고 있다. 그래서, hyper-linked된 API 문서를 만드는 것은 쉽다.

이 매뉴얼은 새로운 releases들과 함께만 업데이트 된다. 소스 컨트롤에 있는 버전은 구식일 가능성이 높다.

## 피드백과 버그 리포팅

만약 너가 Box2D에 대해 질문이나 피드백을 가지고 있다면, 포럼에서 댓글을 남겨라. 이곳은 또한 커뮤니티 토론을 위한 좋은 장소이다.

Box2D는 문제는 Gogle code project를 사용해서 추적된다. 이 곳은 문제를 추적하는 훌륭한 방법이고, 너의 문제가 포럼 깊은 곳에서 잃어버려지지 않도록 보장해준다.

버그와 기능 요구사항을 여기에 쌓아라: <http://code.google.com/p/box2d/>

충분한 세부사항을 제공한다면, 당신의 문제가 고쳐질 가능성을 높이는데 도울 수 있다. 그 문제를 복사한 testbed example이 이상적이다. 너는 이 문서에서 나중에 testbed에 대한 것을 읽을 수 있다.

## 핵심 개념들

Box2D는 몇 가지 기본 개념들과 오브젝트들로 작업한다. 우리는 간단하게 여기에서 이러한 오브젝트들을 정의하고, 좀 더 상세한 세부사항들은 이 문서의 나중에 주어진다.

#### shape

shape는 원 또는 polygon(다각형) 같은 2D(이차원) 기하학 오브젝트이다.

#### rigid body

한 물체(matter)의 덩어리(chunk). 그 덩어리가 매우 딱딱해서(튼튼해서), 그 덩어리에 있는 물질의 어떤 두 개의 부분 사이의 거리가 일정하다. 그것들은 다이아몬드 처럼 딱딱하다. 다음의 토론에서, 우리는 body라는 단어를 rigid body와 혼용해서 사용한다.

#### fixture

fixture는 shape를 body에 bind(결속시키다)하고, 밀도(density), 마찰, 반발(restitution)과 같은 물질적 특성을 더한다. fixture는 shape를 충돌 시스템 (broad-phase)에 넣는데, 그것이 다른 shapes들과 충돌할 수 있도록 하기 위해서이다.

#### Constraint (제약)

constraint는 bodies로부터 나온 자유도(degrees of freedom)을 제거하는 물리적 연결(physical connection)이다. 2D body는 3개의 자유도를 갖는다 (두 개의 translation(평행이동) coordinates(좌표)와 한 개의 회전 좌표). 만약 우리가 한 body를가져와서, 그것을 벽에 고정시킨다면 (시계의 추 처럼), 우리는 그 body를 벽에 constrained(제약)시키는 것이다. 이 점에서, 그 body는 오직 그 pin에 대해서만 회전할 수 있다. 그래서 그 constraint는 2개의 자유도를 제거했다.

#### contact constraint (접촉 제약)

rigid bodies들이 관통하는 것을 방지하고, 마찰과 restitution을 재현하기 위해 설계된 특별한 제약. 너가 contact constraints만들지 않는다. 그것들은 Box2D에 의해 자동적으로 생성된다.

#### Joint (관절)

이것은 두 개 이상의 bodies를 함께 유지하기 위해 사용되는 제약이다. Box2D는 몇 가지 joint types를 지원한다: revolute, prismatic, distance, 그리고 그 이상의 것들. 몇 가지 관절들은 limit과 motors를 갖는다.

#### 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

joint motor는 joint의 자유도에 따라서 연결된 bodies의 움직임을 이끈다. 예를들어, 한 팔꿈치의 회전을 이끌기 위해서 motor를 사용할 수 있다.

#### world

physics world는 서로 상호작용하는 bodies, fixturex, constraints의 집합이다. Box2D는 여러 개의 worlds의 생성을 지원하지만, 이것은 보통 필수적이지 않고 바람직하지 않다.

#### solver

physics world는 시간을 진행시키고 contact와 joint 제약을 해결하는데 사용되는 solver를 갖는다. Box2D solver는 N time에 작동하는 높은 성능의 iterative(반복의) solver이다. 여기에서 N은 제약의 개수이다.

#### continuous collision

solver는 discrete(이산) time steps을 사용하는 시간에서 bodies의 움직임을 진행시킨다. 이것은 어떤 간섭 없이 tunneling을 이끌 수 있다 (아래의 그림에 의해, 오각형과 한 평면이 time0과 time1사이에 충돌해야 하지만, 빠른 속도로 인해 충돌을 하지 않고 넘어가는 상황을 tunneling이라고 부른다).



Box2D는 tunneling을 다루기 위해 특수화된 알고리즘을 포함한다. 첫 번째, 충돌 알고리즘은 충돌의 첫 번째 시간(first Time Of Impact, TOI)를 찾기위해 두 bodies의 움직임을 interpolate(보간)한다. 두 번째, bodies를 그것들의 첫 번째 충돌의 시간을 찾고, 그러고나서 충돌을 해결하는 sub-stepping solver가 있다. (Erin Catto의 2013 GDC 발표 참고, <https://youtu.be/7_nKOET6zwI>)

## Modules

Box2D는 세 개의 모듈로 구성되어 있다: Common, Collision, 그리고 Dynamics. Common module은 할당, 수학, 설정을 위한 코드를 가진다. Collision 모듈은 shapes, broad-phase, collision functions/queries를 정의한다. 마지막으로, Dynamics 모듈은 simulation world, bodies, fixtures, joints를 제공한다.



## Units (단위들)

Box2D는 부동 소수점 숫자로 작동하고, tolerance(허용 오차)는 Box2D가 더 잘 작동하도록 하기 위해 사용되어져야 한다. 이러한 toleracnes는 meters-kilogram-second(MKS) 단위와 함께 잘 작동하도록 조절되어야만 한다. 특히, Box2D는 0.1 ~ 10 meters 사이에서 움직이는 shapes와 잘 작동하도록 설정되어 있다. 그래서 이것은 사이즈 측면에서 soup cans과 버스 사이의 오브젝트들이 잘 작동해야 한다는 것을 의미한다. Static(움직이지 않는) shapes는 문제 없이 50미터 길이까지 될 지도 모른다.

2D physics engine이기 때문에, 픽셀을 단위로 사용하는 것은 유혹적이다. 불행하게도, 이것은 좋지 않은 시뮬레이션과 가급적 이상한 행동들을 이끌 것이다. 길이가 200 픽셀의 한 오브젝트는 Box2D에서, 45층 높이의 빌딩 크기로 보여질 것이다.

**경고**

Box2D는 MKS 단위를 위해 조정되어 있다. 움직이는 오브젝트들의 사이즈를 대강 0.1 ~ 10미터 사이로 유지해라. 너의 환경과 actors를 렌더링할 때, 어떤 scaling system을 사용할 필요가 있을 것이다. Box2D testbed는 OpenGL viewport transform을 사용하여 이것을 한다. "픽셀을 사용하지 마라"

Box2D bodies를 너의 작품을 위에 붙인 움직이는 게시판(billboard)으로 생각하는 것이 가장 좋다. 그 게시판은 미터 단위 시스템에서 움직일지도 모르지만, 너는 그것을 간단한 scaling factor로 픽셀 좌표로 변환할 수 있다. 너는 그러고나서 그러한 픽셀 좌표를 너의 sprites를 배치하는데 사용할 수 있다. 너는 또한 뒤집어진(flipped) 좌표 축을 처리할 수 있다.

Box2D는 각도를 위해 radians를 사용한다. body rotation은 radians로 저장되고, 제한이 없이 커질지도 모른다. 만약 각도의 크기가 너무 커진다면 너의 bodies의 각도를 표준화(normalizing)할 것을 고려해라 (b2Body::SetAngle을 사용해라).

**경고**

Box2D는 radians를 사용한다, 각도(degrees)가 아니라.

## Factories and Definitions

빠른 메모리 관리는 Box2D API의 설계에서 중심 역할을 한다. 그래서 너가 b2Body또는 b2Joint를 생성할 때, 너는 b2World에 있는 factory functions들을 호출할 필요가 있다. 너는 결코 이러한 types들을 다른 방법으로 할당하려고 해선 안된다.

생성 함수들이 있다:

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

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

그리고 대응되는 소멸(파괴, destruction) 함수들이 있다:

void b2World::DestroyBody(b2Body\* body)

void b2World::DestroyJoint(b2Joint\* joint)

한 body또는 joint를 만들려고할 때, 너는 한 정의(definition)를 제공할 필요가 있다. 이러한 정의들은 body또는 joint를 구성하는데 필요한 모든 정보를 포함한다. 이 접근법을 사용하여, 우리는 구성 에러를 방지할 수 있고, 함수 파라미터들의 개수를 작게 유지하고, 분별있는 초기값을 제공하고, 접근자(accessors)의 개수를 줄일 수 있다.

fixtures(shapes)가 body에 부모가 되어야 하기 때문에, 그것들은 b2Body에 있는 factory method를 사용하여 생성되고 파괴된다. :

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

void b2Body::DestroyFixture(b2Fixture\* fixture)

또한 fixture를 shape와 density로부터 직접 생성하는 지름길이 있다.

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

Factories는 그 정의에 대한 참조를 보유하지 않는다. 그래서 너는 스택에 definitions를 생성할 수 있고, 임의의 자원에 그것들을 유지할 수 있다.

# Hello Box2D

Hello World project가 Box2D의 배포판에 있다. 그 프로그램은 큰 ground box와 작은 dynamic box를 만든다. 이 코드는 어떠한 그래픽스를 포함하지 않는다. 너가 보게 될 모든 것은 시간에 따른 박스의 위치에 대한 콘솔에서의 텍스트 결과물이다.

이것은 Box2D로 어떻게 작동하는지에 대한 좋은 예제이다.

## Creating a World

모든 Box2D 프로그램은 b2World 오브젝트의 생성으로 시작한다. b2World는 memory, objects, 그리고 시뮬레이션을 관리하는 physics hub이다. 너는 physics world를 stack, heap 또는 data section에 할당할 수 있다.

Box2D world를 만드는 것은 쉽다. 처음에, 우리는 gravity vector를 정의한다.

b2Vec2 gravity(0.0f, -10.0f);

이제 우리는 world 오브젝트를 생성한다. 우리가 스택에 world를 만들고 있다는 것에 주목해라. 그래서 그 world는 그 scope에 남아있어야 한다.

b2World world(gravity);

그래서 이제 우리는 우리의 physics world를 가지고 있다. 그것에 몇 가지 물건들을 추가해보기 시작하자.

## Creating a Ground Box

Bodies는 다음의 단계들을 사용하여 구성된다:

1. 한 body를 position, damping, 등으로 정의하자.
2. body를 만들기 위해서 world object를 사용해라.
3. shape, 마찰, 밀도 등으로 fixtures를 정의해라.
4. body에 fixtures를 생성해라.

단계 1을위해, 우리는 ground body를 생성한다. 이것을 위해 우리는 body definition이 필요하다. body definition으로, 우리는 groundbody의 초기 위치를 명시한다.

b2BodyDef groundBodyDef;

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

단계 2를 위해, 그 body definition은 ground body를 만들기 위해 world 오브젝트에 넘겨진다. 그 world 오브젝트는 body definition에 대한 참조를 유지하지 않는다. bodies는 기본적으로 static이다. Static bodies는 다른 static bodies와 충돌하지 않고 움직일 수 없다.

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

단계 3을 위해, 우리는 ground polygon을 생성한다. 우리는 ground polygon이 box shape로 만들기 위해 SetAsBox 지름길을 사용한다. 그 box는 parent body의 원점(origin)에 중심을 갖는다.

b2PolygonShape groundBox;

groundBox.SetAsBox(50.0f, 10.0f);

SetAsBox 함수는 half-width(반 너비) 와 half-height(반 높이, extents)를 취한다. 그래서 이 경우에, 그 ground box는 100 단위 너비이고 (x-axis), 20 단위 높이 (y-axis)이다. Box2D는 미터, 킬로그램, 그리고 초에 대해 조정되어 있다. 그래서 너는 그 길이들이 (extents) 미터 단위에 있다고 고려할 수 있다. Box2D는 일반적으로 오브젝트들이 일반적인 실제 세계의 오브젝트들의 크기일 때 가장 잘 작동한다. 예를들어, 한 통(barrel)은 1미터 높이이다. 부동소수점 연산의 한계 때문에, 빙하 또는 먼지 파티클의 움직임을 모델링하기 위해 Box2D를 사용하는 것은 좋은 아이디어가 아니다.

우리는 shape fixture를 생성하여 step 4에서 ground body를 마무리한다. 이 단계를 위해서, 우리는 지름길을 사용한다. 우리는 default fixture material properties를 바꿀 필요가 없다. 그래서 우리는 fixture definition을 생성하지 않고 그 shape를 직접 body에 넘길 수 있다. 나중에 우리는 customized된 material properties를 위한 fixture definition을어떻게 사용할지를 볼 것이다. static body는 정의 상 zero mass를 갖는다. 그래서 그 밀도(density)는 이 경우에 사용되지 않는다.

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

Box2D는 그 shape에 대한 참조를 유지하지 않는다. 그것은 그 데이터를 새로운 b2Shape 오브젝트에 복사한다.

모든 fixture가 parent body를 가져야 한다는 것에 주목해라. 심지어 static인 fixtures들도. 그러나, 너는 모든 static fixtures들을 단일 static body에 붙일 수 있다.

너가 fixture를 사용하여 한 shape를 body에 붙일 때, 그 shape의 좌표는 그 body에 local이 된다. 그래서 그 body가 움직일 때, shape도 움직인다. fixture의 world transform은 그 parent body로부터 상속된다. fixture는 body와 독립적인 transform을 갖지 않는다. 그래서 우리는 body에서 shape를 움직이지 않는다. 한 body에 있는 shape를 움직이게하거나 수정하는 것은 지원되지 않는다. 그 이유는 간단하다: 변하는 shapes를 가진 body는 rigid body가 아니다. 그러나 Box2D는 rigid body engine이다. Box2D에서 만들어지는 많은 가정들은 rigid body model에 기반을 둔다. 만약 이것이 위반된다면, 많은 것들이 고장 날 것이다.

## Creating a Dynamic Body

그래서 우리는 이제 ground body를 가진다. 우리는 dynamic body를만들기위해 같은 기법을사용할 수 있다. 크기 외에, 주된 차이점은 우리가 dynamic body의 mass properties를 설정해야한다는 것이다.

처음에 우리는 CreateBody를 사용하여 그 body를 생성한다. 기본적으로 bodies는 static이다. 그래서 우리는 생성 시간에 그 body를 dynamic으로 만들기 위해 b2BodyType을 설정해야만 한다.

b2BodyDef bodyDef;

bodyDef.type = b2\_dynamicBody;

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

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

경고

너는 body가 forces에 반응하여 움직이기를 원한다면 body type을 b2\_dynamicBody로 설정해야 한다.

다음으로 우리는 fixture definition을 사용하여 polygon shape를 생성하고 붙인다. 처음에 우리는 box shape를 만든다.:

b2PolygonShape dynamicBox;

dynamicBox.SetAsBox(1.0f, 1.0f);

다음으로 우리는 box를 사용하여 fixture definition을 생성한다. 우리가 밀도를 1로 설정한 것에 주의해라. 기본 density는 zero이다. 또한, shape에 대한 마찰은 0.3으로 설정된다.

b2FixtureDef fixtureDef;

fixtureDef.shape = &dynamicBox;

fixtureDef.density = 1.0f;

fixtureDef.friction = 0.3f;

경고

dynaimc body는 적어도 0이 아닌 density를 가진 한 fixture를 가져야 한다. 그렇지 않다면, 너는 이상한 행동을 얻을 것이다.

fixture definition을 사용하여, 우리는 그 fixture를 이제 생성할 수 있다. 이것은 body의 질량을 자동으로 업데이트 한다. 너는 너가 원하는 만큼 body에 많은 fixtures를 더할 수 있다. 각각은 total mass에 기여한다

body->CreateFixture(&fixtureDef);

이것이 초기화를 위한 것이다. 우리는 이제 시뮬레이션을 시작할 준비가 됐다.

## Simulating the World (of Box2D)

그래서 우리는 ground box와 dynamic box를 초기화했다. 이제 우리는 Newton을 느슨하게 설정할 준비가 되어있다. 그의 작업을 하기 위해서. 우리는 고려해야할 두 개의 더 많은 이슈들을 가지고 있다.

Box2D는 integrator라고 불려지는 computational algorithm을 사용한다. Integrators는 이산 시간에 (discrete points of time) 물리 방정식을 재현한다. 이것은 전통적인 게임 loop를 따라 가는데, 거기에서 우리는 필수적으로 스크린에서 움직임의 flip book을 갖게된다 (책을 넘기면서 움직임을 재현하는 과정). 그래서 우리는 Box2D의 time step을 고를 필요가 있다. 일반적으로 게임을 위한 물리엔진들은 60Hz 즉 1/60 초만큼 적어도 빠른 time step을 좋아한다. 너는 더 큰 time step으로 할 수 있지만, 너의 world를 위해 그 정의를 설정하는 것에 대해 좀 더 세심해야 할 것이다. 우리는 또한 많이 변하는 time step을 좋아하지 않는다. 가변 time step은 변할 수 있는 결과를 만들어낸다. 그리고 이것은 디버깅하는 것을 어렵게 만든다. 그래서 time step을 너의 프레임율에 고정하지 말아라 (만약 너가 정말 그렇게 못한다면, 정말 그렇게 해야한다). 더 이상 덧붙이지 않고, 여기에 그 time step이 있다.

float32 timeStep = 1.0f / 60.0f;

integrator외에도, Box2D는 constraint solver라고 불려지는 더 큰 코드를 사용한다. constraint solver는 시뮬레이션에서 한 time에 한 개 씩 모든 constraints를 해결한다. single constraint는 완벽하게 해결될 수 있다. 그러나, 우리가 한 개의 constraint를 해결할 때, 우리는 약간 다른 constraints를 방해한다. 좋은 솔루션을 얻기 이ㅜ해, 우리는 모든 constraints에 대해 여러 번 반복할 필요가 있다.

constraint solver에서 두 단계가 있다: veloctiy phase와 position phase. velocityphase에서, 그 solver는 bodies를 정확히 움직이게하는데 필요한 impulse(충격량)을 연산한다. position phase에서, solver는 overlap(중첩)과 joint detachment(관절 분리)를 제거하기 위해 bodies의 위치를 조절한다. 각 phase는 그것 자신의 iteration count를 갖는다. 게다가, position phase는 에러가 잘다면 반복을 일찍 끝낼지도 모른다.

Box2D를 위해 제안되는 반복 횟수는 velocity에 대해서는 8이고 position에 대해서는 3이다. 너는 이 숫자를 너의 선호에 따라 조절할 수 있고, 이것이 성능과 정확성 사이의 trade-off(거래)라는 것을 명심해야 한다. 더 적은 반복을 사용하는 것은 성능을 증가시키지만, 정확성이 피해를 입는다. 마찬가지로, 더 많은 반복을 사용하는 것은 성능을 줄이지만, 너의 시뮬레이션 퀄리티를 향상시킨다. 이 간단한 예제에 대해서, 우리는 더 많은 반복이 필요하지 않다. 여기에 우리가 선택한 반복 횟수가 있다.

int32 velocityIterations = 6;

int32 positionIterations = 2;

time step과 iteration count가 완전히 관계없다는 것에 주목해라. iteration은 sub-step이 아니다. 한 solver iteration은 한 time step 내에서, 모든 constraints에 대해 single pass이다. 너는 단일 time step 내에서, constraints에 대해 여러 개의 passes를 가질 수 있다.

우리는 이제 simulation loop를 시작할 준비가 되어있다. 너의 게임에서, simulation loop는 너의 게임 loop와 통합되어질 수 있다. 너의 게임 loop를 통과하는 각 pass에서, 너는 b2World::Step을 호출한다. 너의 프레임율과 physics timestep에 따라, 한 번의 호출이 보통 충분하다.

그 Hello World program은 간단하게 되도록 설계되었다. 그래서 그것은 어떠한 그래픽 결과물을 갖지 않는다. 그 코드는 dynamic body의 위치와 회전을 출력한다. 여기에 총 1초의 재현된 시간에 대해 60 time steps를 재현한 시뮬레이션 loop가 있다.

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

{

world.Step(timeStep, velocityIterations, positionIterations);

b2Vec2 position = body->GetPosition();

float32 angle = body->GetAngle();

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

}

그 결과물은 box가 ground box에 떨어지고 착륙하는 것을 보여준다. 너의 output은 이것처럼 보여야 한다:

0.00 4.00 0.00

0.00 3.99 0.00

0.00 3.98 0.00

...

0.00 1.25 0.00

0.00 1.13 0.00

0.00 1.01 0.00

## Cleanup

한 world가 scope를 떠나거나 한 포인터에 대해 delete를 호출하여 삭제될 때, bodies, fixtures, joints에 대해 보유된 모든 메모리는 해제된다. 이것은 성능을 향상하고 너를 더욱 편리하게 하도록 처리된다. 그러나, 너가 가진 어떤 body, fixture, 또는 joint pointers에 null 값을 넣을 필요가 있을 것이다. 왜냐하면 그것들이 유효하지 않을 것이기 때문이다.

## The Testbed

일단 너가 HellowWorld example을 정복했다면, 너는 Box2D의 testbed를 보기 시작해야만 한다. 그 testbed는 단위 테스트 framework이고 demo 환경이다. 여기에 몇 가지 특징들이 있다:

* Pan(보여주고) zoom하는 카메라
* Dynamic bodies에 부착된 shapes에 대한 Mouse Picking
* 확장가능한 테스트 셋
* 테스트, parameter tuning, debug drawing options를 고르는 GUI
* 일시정지와 단일 step simulation
* Text rendering.



그 tesetbed는 테스트 케이스와 그 프레임워크 자페에서 Box2D 사용의 많은 예제들을 가지고 있다. 나는 너가 Box2D를 배움에 따라 testbed로 탐험하고 가지고 노는 것을 격려한다.

주의 : testbed는 freeglut과 GLUI를 사용하여 쓰였다. 그 testbed는 Box2D 라이브러리의 일부가 아니다. Box2D 라이브러리는 렌더링에 대해서 모른다. HelloWorld example에 의해 보여지듯이, 너는 Box2D를 사용하기 위해 renderer가 필요하지 않다.

# Common

## About

Common 모듈은 설정, 메모리 관리, 그리고 벡터 수학을 포함한다.

## Settings

b2Setting.h 헤더는 다음을 포함한다:

* int32와 float32같은 Types들
* 상수
* 할당 wrappers
* The version number

### Types

Box2D는 구조체의 크기를 결정하는 것을 쉽게하기 위해 float32, int8 등등과 같은 다양한 types들을정의한다.

### Constants

Box2D는 몇 가지 상수들을 정의한다. 이러한 것들은 모두 b2Setting.h에 문서화되어 있다. 일반적으로 너는 이러한 상수들을 조절할 필요가 없다.

Box2D는 충돌과 시뮬레이션을 위해 부동 소수점 수학을 사용한다. round-off error(반올림 에러)에 의해, 몇 가지 숫자와 관련된 tolerances가 정이된다. 몇 가지 tolerances들은 절대적이고, 어떤 것은 상대적이다. Absolute(절대적) tolerance는 MKS 단위를 사용한다.

### Allocation wrappers

그 settings file은 큰 할당을 위해 b2Alloc과 b2Free를 정의한다. 너는 이러한 호출들을 너 자신의 메모리 관리 시스템에 보낼지도 모른다.

### Version

b2Version 구조체는 현재 버전을 가지고 있다. 그래서 너는 런타임에 이것을 쿼리(query)할 수 있다.

## Memory Management

Box2D의 설계에 대한 많은 결정들은 빠르고 효율적인 메모리 사용을 위한 필요성에 기반을 둔다. 이 섹션에서,나는 Box2D가 메모리를 어떻게 그리고 왜 할당하는지를 이야기할 것이다.

Box2D는 많은 수의 작은 오브젝트들 (약 50~300 바이트)을 할당하는 경향이 있다. 작은 오브젝트들에 대해 malloc 또는 new를 통해 system heap을 사용하는 것은 비효율적이고, 파편화를 발생시킬 수 있다. 이러한 작은 오브젝트들은 짧은 수명을 가질지도 모른다, contacts 같이, 그러나 몇 개의 time steps동안 지속할 수 있다. 그래서 우리는 효율적으로 이러한 오브젝트들에 대해 heap memory를 제공할 수 있는 allocator(할당자)가 필요하다.

Box2D의 솔루션은 b2BlockAllocator라고 불리는 small object allocator(SOA)를 사용하는 것이다. SOA는 가변 크기의 성장가능한(growable) 많은 pools을 유지한다. 메모리에 대한 요청이 있을 때, SOA는 그 요청된 사이즈에 가장 맞는 메모리 블럭을 반환한다. 한 block이 해제될 때, 그것은 pool에 반환된다. 이러한 연산들 둘 다 빠르고 거의 heap traffic을 일으키지 않는다.

Box2D는 SOA를 사용하기 때문에, 너는 결코 body, fixture, joint를 new or malloc해선 안된다. 그러나, 너는 b2World를 너 스스로 할당해야만 한다. 그 b2World 클래스는 bodies, fixtures, and joints를 너가 생성하도록 하는 factories를 제공한다. 이것은 Box2D가 SOA를 사용하고, 복잡한 세부사항을 너로부터 숨기게 해준다. 결코 body, fixture,joint에 대해 delete나 free를 호출하지 말아라.

한 time step을 실행하는 동안, Box2D는 임의의 workspace memory가 필요하다. 이것을 위해서, 그것은 b2StackAllocator라고 불리는 stack allocator를 사용한다. step마다의 heap allocations을 피하기 위해서이다. 너는 stack allocator와 상호작용 할 필요가 없다. 그러나 그것이 있다는 것을 아는 것은 좋다.

## Math

Box2D는 간단한 작은 vector and matrix 모듈을 포함한다. 이것은 Box2D와 그 API의 내부 필요성에 맞도록 설계되었다. 모든 멤버들은 노출되어있다. 그래서 너는 그것들을 너의 프로그램에서 자유롭게 사용해도 된다.

그 수학 라이브러리는 Box2D가 포팅하고 유지되기 쉽도록 만들기 위해 간단하게 유지된다.

# Collision Module

## About

Collision 모듈은 shapes를 포함하고, 그것들에 작동하는 함수들을 포함한다. 그 모듈은 또한 큰 시스템의 충돌 처리를 가속하기 위해 dynamic tree와 broad-phase를 포함한다.

충돌 모듈은 dynamic system의 바깥에서 사용할 수 있도록 설계된다. 예를들어, 너는 물리외에 너의 게임의 다른 측면을 위해 dynamic tree를 사용할 수 있다.

그러나, Box2D의 주된 목적은 rigid body physics engine을 제공하는 것이다. 그래서 collision module을 그 자체로 사용하는 것은 어떤 프로그램에 대해 제한됨을 느낄 것이다. 마찬가지로, 나는 그것을 문서화하거나 그 API를 정리할 많은 노력을 하지 않을 것이다.

## Shapes

Shapes는 collision geometry(충돌 도형)을 묘사하고, 물리 시뮬레이션과 별개로 사용된다. 최소한으로, 너는 rigid bodies에 나중에 부착될 shapes를 어떻게 생성하는지를 이해해야 한다.

Box2D shapes는 b2Shape base class를 구현한다. 그 base class는 다음을 하기 위해 함수들을 정의한다:

* Shape로 중첩(overlap)에 대해 한 점을 테스트
* Shape에 대해 ray cast를 수행
* Shape의 AABB를 연산
* Shape의 mass properties를 연산

게다가, 각 shape는 type member와 radius를 갖는다. 그 radius는 아래에서 이야기 되듯이 심지어 polygons에 적용된다.

한 shape가 bodies에 대해 모르고, dynamics system과 별개로 있다는 것을 명심해라. Shapes는 크기와 성능을 위해 최적화된 compact form에 저장되어 있다. 그러하듯이, shapes는 쉽게 이동되지 않는다. 너는 한 shape를 움직이게 하기 위해 수동으로 shape vertex positions을 설정해야 한다. 그러나, 한 shape가 fixture를 사용하여 body에 부착될 떄, 그 shapes는 host body와 함께 좀 더 rigidly하게 움직인다. 요약해서:

* shape가 한 body에 부착되지 않을 때, 너는 그것이 world-space에서 표현되는 정점들로 볼 수 있다.
* 한 shape가 body에 부착될 때, 너는 그것이 local coordinates에서 표현되는 정점들로 볼 수 있다.

### Circle Shapes

Circle shapes는 position과 radius를 갖는다. Circles은 solid하다. 너는 circle shape를 사용하여 텅빈 circle을 만들 수 없다.

b2CircleShape circle;

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

circle.m\_radius = 0.5f;

### Polygon Shapes

Polygon shapes는 solid convex polygons이다. polygon은 그 내부에 있는 두 점을 연결하는 모든 직선들이 polygon의 어떠한 edge(변)을 가로지르지 않을 때 convex(볼록)하다. Polygons는 solid하고 결코 텅 비어있지 않다(hollow). 한 polygon은 3개 이상의 정점을 가져야만 한다.



Polygon 정점들은 반 시계 방향(CCW)로 저장된다. 우리는 조심해야 하는데, CCW의 개념은 평면의 밖을 가리키는 Z축을 가진 오른손 좌표계와 관련이 있기 때문이다.

그 polygon members는 public이지만, 너는 polygon을 생성하기 위해 초기화 함수들을 사용해야 한다. 그 초기화 함수들은 normal(법선) vectors을 생성하고, validation(유효화)를 수행한다.

너는 한 정점 배열을 넘겨서 polygon shape를 생성할 수 있다. 그 배열의 최대 크기는 b2\_maxPolygonVertices에 의해 통제되고, 그것은 8의 디폴트 값을 가진다. 이것은 가장 convex한 polygons을 묘사하는데 충분하다.

b2PolygonShape::Set 함수는 자동으로 그 convex hull를 연산하고, 적절한 winding(감기는) order(순서)를 설정한다. 이 함수는 정점의 개수가 적을 때 빠르다. 만약 너가 b2\_maxPolygonVertices를 증가시킨다면, 그러면 그 convex hull computation는 느려질지도 모른다. 또한 convex hull 함수가 너가 제공한 정점들을 제거하고/또는 재순서(re-order)시킬지도 모른다는 것에 주목해라. b2\_linearSlop보다 더 가까운 정점들은 합쳐질지도 모른다.

// 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);

그 polygon shape는 boxes를 만드는 몇 가지 편의성(convenience) 함수들을 가지고 있다.

void SetAsBox(float32 hx, float32 hy);

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

Polygons은 b2Shape로부터 radius를 상속받는다. 그 radius는 polygon을 둘러싸는 skin을 생성한다. 그 skin은 stacking scenarios(물건을 쌓는 상황)에서 polygons를 조금 분리시키기 위해서 사용된다. 이것은 core polygon에 대해 continuous collision이 작동하도록 한다.



그 polygon skin은 polygons이 분리되게 하여 tunneling을 방지하는 것을 도와준다. 너의 시각적 representation(표기)은 어떤 간격을 숨기기 위해 그 polygon보다 더 크게 될 수 있다.



### Edge Shapes

Edge Shapes는 선분이다. 이러한 것은 너의 게임을 위해 free-form static environment를 만드는 것을 돕기위해 제공된다. edge shapes의 중요한 한계는 그것들이 circles와 polygons와 충돌할 수 있지만, 그들 자신끼리 할 수 없다는 것이다. Box2D에 의해 사용되는 중요한 알고리즘들은 두 개의 충돌하는 shapes중의 하나가 부피(volume)를 갖기를 요구한다. Edge shapes는 어떠한 volume을 가지고 있지 않다. 그래서 edge-edge collision은 가능하지 않다.

// This an edge shape.

b2Vec2 v1(0.0f, 0.0f);

b2Vec2 v2(1.0f, 0.0f);

b2EdgeShape edge;

edge.Set(v1, v2);

많은 경우에, 게임 환경은 몇 가지 edge shapes를 끝에서 끝으로 연결하여 구성된다. 이것은 한 polygon이 edges의 chain을 따라 미끄러질 때 예상치않은 artifact를 발생시킬 수 있다. 아래의 그림에서, 우리는 내부 vertex와 충돌하는 한 box를 본다. polygon이 internal collision normal을 생성하는 internal vertex와 충돌할 때 이러한 ghost 충돌이 발생한다.



만약 edg1이 존재하지 않았다면, 이 충돌은 괜찮은 것처럼 보일 것이다. edge1이 존재하여, 그 internal collision은 버그처럼 보일것이다. 그러나 일반적으로 Box2D는 두 개의 shapes를 충돌시킬 때, Box2D는 그것들을 고립되게 본다.

운 좋게도, 그 edge shape는 인접 ghost vertices를 저장하여 ghost collisions를 제거하는 메커니즘을 제공한다. Box2D는 내부 충돌을 방지하기 위해 이러한 ghost vertices를 사용한다.



// This is an edge shape with ghost vertices.

b2Vec2 v0(1.7f, 0.0f);

b2Vec2 v1(1.0f, 0.25f);

b2Vec2 v2(0.0f, 0.0f);

b2Vec2 v3(-1.7f, 0.4f);

b2EdgeShape edge;

edge.Set(v1, v2);

edge.m\_hasVertex0 = true;

edge.m\_hasVertex3 = true;

edge.m\_vertex0 = v0;

edge.m\_vertex3 = v3;

일반적으로 이 방식으로 edges들을 함께 연결하는 것은 낭비적이고 지루하다. 이것은 우리를 chain shapes로 데려간다.

### Chain Shapes

chain shape는 너의 staticgame worlds를 구성하기위해 많은 edges를 함께 연결하는 효율적인 방법을 제공한다. Chain shapes는 자동으로 ghost collisions을 제거하고, two-sided(양면) collisions을 제공한다.



// This a chain shape with isolated vertices

b2Vec2 vs[4];

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

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

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

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

b2ChainShape chain;

chain.CreateChain(vs, 4);

너는 스크롤 게임 world를 가질지 모르고, 몇 가지 chains을 함께 연결하고 싶어한다. 너는 ghost vertices를 사용하여 함께 chain을 연결할 수 있다. 우리가 b2EdgeShape에서 했던 것 처럼.

// Install ghost vertices

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

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

너는 또한 loops를 자동으로 만들어도 된다.

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

b2ChainShape chain;

chain.CreateLoop(vs, 4);

chain shapes의 자체 교차 테스트(self-intersection)는 지원되지 않는다. 그것은 작동할지도 안할지도 모른다. ghost collision을 방지하는 코드는 그 chain에 self-intersections이 없다고 가정한다. 또한, 매우 가까운 정점들은 문제를 발생시킬 수 있다. 모든 너의 edges가 b2\_linearSlop(5mm)보다 더 길도록 해라.



chain에 있는 각 edge는 child shape로서 다뤄지고, index로 접근될 수 있다. 한 chain shape가 body에 연결될 때, 각 edge는 broad phase collision tree에서 그것 자신의 bounding box를 얻게된다.

// Visit each child edge.

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

{

b2EdgeShape edge;

chain.GetChildEdge(&edge, i);

…

}

## Unary Geometric Queries

너는 single shape에 대해 두 개의 geometric queries를 수행할 수 있다.

### Shape Point Test

너는 한 shape로 overlap에 대해 한 점을 테스트할 수 있다. 너는 그 shape에 대한 transform을제공하고, world point를 제공한다.

b2Transfrom transform;

transform.SetIdentity();

b2Vec2 point(5.0f, 2.0f);

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

Edge와 chain shapes는 항상 false를 반환한다, 비록 그 chain이 loop일지라도.

### Shape Ray Cast

너는 첫 번째 교차점과 normal vector를 얻기위해 한 shape에 ray를 날릴 수 있다. 만약 그 ray가 그 shape안에서 시작한다면, 어떠한 hit도 등록되지 않을 것이다. chain shapes에 대해서는 child index가 포함되는데, 그 ray cast가 오직 한 번에 single edge만을 체크할 것이기 때문이다.

b2Transfrom transform;

transform.SetIdentity();

b2RayCastInput input;

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

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

input.maxFraction = 1.0f;

int32 childIndex = 0;

b2RayCastOutput output;

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

if (hit)

{

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

…

}

## Binary Functions

충돌 모듈은 shapes의 한 쌍을 취하고 몇 가지 결과를 연산하는 bilateral functions를 포함한다. 이러한 것은 다음의 것을 포함한다:

* Overlap (중첩)
* Contact manifolds (충돌 시 생기는 접촉점의 정보)
* Distance (거리)
* Time of impact (충돌 시간)

### Overlap

너는 이 함수를 사용해서 overlap에대해 두 shapes를 테스트할 수 있다:

b2Transform xfA = …, xfB = …;

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

다시, 너는 chain shapes의 경우에 child indices를 제공해야만 한다.

### Contact Manifolds

Box2D는 중첩하는 shapes에 대해 contact points(접촉점)을 연산하는 함수를 가지고 있다. 만약 우리가 circle-circle 또는 circle-polygon을 고려한다면, 우리는 오직 한 contact와 normal만을 얻을 수 있다. 이러한 점들은 같은 normal vector를 공유한다. 그래서 Box2D는 그것들을 manifold 구조체로 그룹짓는다. 그 contact solver는 stacking stability(쌓는 것의 안정성)을 향상시키기 위해 이것을 이용한다.



보통 너는 contact manifolds를 직접적으로 연산할 필요가 없다. 그러나 너는 그 시뮬레이션에서 만들어진 결과들을 사용할 가능성이 크다.

그 b2Manifold 구조체는 normal vector를 보유하고, 두 개 까지의 contact points를 보유한다. 그 normal과 points는 local coordinates에서 보유된다. 그 contact solver의 편리성으로서, 각 점은 normal과 tangential (friction) impulses를 저장한다.

b2Manifold에 저장된 data는 내부 사용을 위해 최적화된다. 만약 너가 이 데이터가 필요하다면, 그 contact normal과 points의 world coordinates를 생성하기 위해 b2WorldManifold 구조체를 사용하는 것이 보통 가장 좋다. 너는 b2Manifold와 shape에게 transforms와 radii를 제공할 필요가 있다.

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];

…

}

world manifold가 원래의 manifold로부터 point count를 사용한다는것에 주의해라.

시뮬레이션 동안, shapes는 움직일지도 모르고, manifolds가 변할지도 모른다. 점들은 추가되거나 제거될지도 모른다. 너는 이것을 b2GetPointStates를 사용해서 탐지할 수 있다.

b2PointState state1[2], state2[2];

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

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

{

// process event

}

### Distance

b2Distance 함수는 두 shapes사이의 거리를 연산하기 위해 사용될 수 있다. 그 distance 함수는 b2DistanceProxy로 변환될 두 개의 shapes가 필요하다. 또한 반복되는 호출에 대해 distance funciton을 warm start하기 위해 사용되는 몇 가지 캐싱이 있다. 너는 b2Distance.h에서 세부사항을 볼 수 있다.



### Time of Impact

만약 두 shapes가 빠르게 움직이고 있다면, 그것들은 single time step안에 서로를 통과(tunnel)해버릴지도 모른다.



그 b2TimeOfImpact 함수는 두 개의 움직이는 shapes가 충돌하는 시간을 결정하기 위해 사용된다. 이것은 time of impact(TOI)라고 불려진다. b2TimeOfImpact의 주된 목적은 tunnel을 방지하는 것이다. 특히, 움직이는 오브젝트가 static level geometry 밖으로 tunneling하는 것을 방지하기 위해 설계되었다.

이 함수는 두 shapes의 rotation과 translation을 설명하지만, 만약그 회전이 충분히 크다면, 그러면 그 함수는 충돌을 놓칠지도 모른다. 그러나 그 함수는 여전히 중첩되지 않은(non-overlapped) 시간을 보ㅗ할 것이고, 모든 translational(평행이동의) 충돌을 포착할 것이다.

그 time of impact 함수는 초기 separating axis(분리축)을 확인하고, 그 shapes가 그 축을 가로지르지 않는지를 보장한다. 이것은 최종 위치에 명백한 충돌을 놓칠지도 모른다. 이 접근법이 몇 가지 충돌을 놓칠지라도, 그것은 tunnel 방지에 매우 빠르고 적절하다.





rotation 크기에 제한을 두는 것은 어렵다. 충돌이 작은 회전에 대해 놓쳐지는 경우들이 있을지도 모른다. 일반적으로, 이러한 놓쳐진 회전 충돌들은 게임 플레이에 피해를 끼치지 않는다. 그것들은 glancing(흘끗보는, 신경안쓸) collision인 경향이 있다.

그 함수는 두 shapes와 (b2DistanceProxy로 변환될) b2Sweep 구조체를 요구한다. 그 sweep 구조체는 shapes의 초기와 최종 transforms을 정의한다.

너는 shape cast를 수행하기 위해 고정된 rotations을 사용할수 있다. 이 경우에, time of impact 함수는 어떤 충돌을 놓칠 것이다.

## Dynamic Tree

b2DynamicTree class는 많은 수의 shapes를 효율적으로 구성하기 위해 Box2D에서 사용된다. 그 클래스는 shapes에 대해 모른다. 대신에, 그것은 user data pointers와 함께 axis-aligned bounding boxes(AABB)에 작동한다.

dynamic tree는 hierarchical AABB tree이다. 그 트리에 있는 각 내부 노드는 두 개의 자식을 갖는다. leaf node는 single user AABB이다. 그 트리는 그 트리를 균형되게 유지하기 위해 rotations을 사용한다. 심지어 degenerate input의 경우에도.

tree 구조체는 효율적인 ray casts와 region(영역) queries를 허용한다. 예를들어, 너는 너의 scene에 수백개의 shapes를 가질지도 모른다. 너는 each shape에 ray casting하여 brute force 방식으로 scene에대해 ray cast를 수행할 수 있다. 이것은 비효율적인데, 왜냐하면 그것은 널리 퍼져있는 shapes를 이용하지 않기 때문이다. 대신에, 너는 dynamic tree를 유지하고, 그 트리에 대해 ray casts를 수행할 수 있다. 이것은 많은 수의 shapes를 생략하고 tree를 통해 ray를 이동시킨다.

region query는 query AABB와 중첩되는 모든 leaf AABB들을 찾기 위해 트리를 사용한다. 이것은 많은 shapes들이 생략되기 때문에 brute force 접근법보다 더 빠르다.





일반적으로, 너는 dynamic tree를 직접적으로 사용하지 않을 것이다. 오히려 너는 ray casts와 region queries를 위해 b2World 클래스를 사용할 것이다. 만약 너가 너만의 dynamic tree를 인스턴스화 할려고 한다면,너는 Box2D가 그것을 어떻게 사용하는지를 보아서 그것을 사용하는 법을 배울 수 있다.

## Broad-phase

한 physics step에서, 충돌 처리는 narrow-phase와 broad-phase로 나눠질 수 있다. narrow-phase에서, 우리는 shapes의 쌍들에 대해 contact points를 연산한다. 우리가 N개의 shapes를 가진다고 상상해라. brute force를 사용하여, 우리는 N \* N / 2 쌍에 대해 narrow-phase를 수행할 필요가 있을 것이다.

b2BroadPhase 클래스는 pair management에 대해 dynamic tree를 사용하여 이 부담을 줄인다. 이것은 크게 narrow-phase calls의 개수를 줄인다.

일반적으로 너는 broad-phase와 직접적으로 상호작용하지 않는다. 대신에, Box2D는 내부적으로 broad-phase를 만들고 관리한다. 또한, b2BroadPhase는 Box2D의 시뮬레이션 loop와 함께 설계되어있다. 그래서 그것은 다른 use cases에 대해 적절하지 않을 가능성이 높다.

# Dynamics Module

## Overview

Dynamics 모듈은 Box2D의 가장 복잡한 부분이고, 너가 가장 상호작용할 가능성이 높은 부분이다. Dynamics 모듈은 Common과 Collision 모듈의 위에 있다. 그래서 너는 어느정도 이제 그것들과 친숙해야만 한다.

Dynamics 모듈은 다음을 포함한다:

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

이러한 클래스 사이에 많은 의존성들이 있다. 그래서 다른 것을 언급하지 않고 한 클래스를 설명하는 것은 어렵다. 다음에서, 너는 아직 설명되지 않은 클래스에 대해 어떤 references들을 볼지도 모른다. 그러므로, 너는 그것을 자세히 읽기전에 이 챕터를 빠르게 생략하기를 원할지도 모른다.

dynamics 모듈은 다음 챕터에서 다뤄진다.

# Bodies

## About

Bodies는 position과 velocity를 가진다. 너는 forces, torques, 그리고 impulses를 bodies에 적용할 수 있다. Bodies는 static, kinematic, or dynamic일 수 있다. 여기에 body type 정의들이 있다:

#### b2\_staticBody

static body는 시뮬레이션에서 움직이지 않고, 무한한 질량을 가진 것처럼 행동한다. 내부적으로, Box2D는 그 질량과 inverse mass를 0으로 저장한다. static bodies는 사용자에 의해 수동으로 움직여질 수 있다. static body는 zero velocity를 가진다. Static bodies는 다른 static 또는 kinematic bodies와 충돌하지 않는다.

#### b2\_kinematicBody

kinematic body는 그것의 velocity에 따라 시뮬레이션에서 움직인다. kinematic bodies는 forces에 반응하지 않는다. 그것들은 사용자에 의해 수동으로 움직여질 수 있다. 그러나, 일반적으로 kinematic body는 그것의 velocity를 설정하여 움직여진다. kinematic body는 마치 그것이 무한의 질량을 가진 것처럼 행동한다. 그러나, Box2D는 mass와 inverse mass에 대해 zero를 저장한다. Kinematic bodies는 다른 kinematic or static bodies와 충돌하지 않는다.

#### b2\_dynamicBody

dynamic body는 완전히 시뮬레이션된다. 그것들은 사용자에 의해 수동으로 움직여질 수 있지만, 일반적으로 forces를 따라서 움직인다. dynamic body는 모든 body types와 충돌한다. dynamic body는 항상 유한의, 0이 아닌 mass를 갖는다. 만약 너가 dynamic body의 mass를 zero로 설정하려고 한다면, 그것은 자동으로 1 kilogram의 질량을 얻을 것이고, 그것은 회전하지 않을 것이다.

Bodies는 fixtures (shapes)에 척추이다. Bodies는 fixtures를 가지고 다니고, world에서 그것들을 움직인다. Bodies는 항상 Box2D에서 rigid bodies이다. 그것은 같은 rigid body에 부착된 두 개의 fixtures는 결코 서로를 기준으로 움직이지 않고, 같은 body에 부착된 fixtures들은 충돌하지 않는다는 것을 의미한다.

Fixtures는 collision geometry와 density를 가지고 있다. 일반적으로 bodies는 fixtures로부터 그것들의 mass properties를 얻는다. 그러나, 너는 body가 생성되고나서 mass properties를 덮어 씌울 수 있다.

너는 보통 너가 생성한 모든 bodies에 대해 포인터들을 유지한다. 이 방식으로, 너는 너의 graphical entities의 positions를 업데이트하기위해 body positions를 query할 수 있다. 너는 또한 body pointers를 유지해야한다. 그래서 너는 너가 그것들로 작업하는게 끝났을 때 그것들을 파괴할 수 있다.

## Body Definition

한 body가 생성되기 전에, 너는 body definition (b2BodyDef)를 생성해야만 한다. 이 body definition은 한 body를 생성하고 초기화하는데 필요한 data를 가지고 있다.

Box2D는 body definition에서 데이터를 카피한다; 그것은 그 body body definition에 대한 pointer를 유지하지 않는다. 이것은 너가 여러 bodies를 생성하기 위해 한 body definition을 재활용할 수 있다는 것을 의미한다.

body definition의 주요 members들을 봐보자.

### Body Type

이 챕터의 시작에서 이야기 되었듯이, 세 개의 다른 body types가 있다: static, kinematic, and dynamic. 너는 그 body type을 생성할 때 설정해야만 한다. 왜냐하면 그 body type을 나중에 바꾸는 것은 비싸기 때문이다.

bodyDef.type = b2\_dynamicBody;

body type을 설정하는 것은 의무적이다.

### Position and Angle

body definition은 너에게 body의 positions을 생성할 때 초기화할 기회를 준다. 이것은 body를 world origin에서 생성하고 그러고나서 움직이는 것 보다 훨씬 더 좋은 성능을 가진다.

**경고**

한 body를 origin에서 생성하고나서 움직이지 마라. 만약 너가 몇 가지 bodies를 origin에서 생성한다면, 성능에 피해를 입을 것이다.

body는 두 개의 흥미로운 주된 점들(points)이 있다. 첫 번째 점은 body의 원점(origin)이다. Fixtures과 joints는 body의 원점을 기준으로 부착된다. 두 번째 흥미로운 점은 mass의 center이다. center of mass는 부착된 shapes의 mass distribution(질량 분산)으로부터 결정되거나 b2MassData로 explicitly하게 설정된다. Box2D의 내부 연산의 대부분은 center of mass position을 사용한다. 예를들어, b2Body는 center of mass에 대한 linear velocity를 저장한다.

너가 body definition을 구성할 때, 너는 center of mass가 어디에 위치한지를 모를수도 있다. 그러므로 너는 그 body의 원점의 position을 명시한다. 너는 또한 그 body의 angle을 radians로 명시할지도 모른다. 그 각은 center of mass의 position에 영향 받지 않는다. 만약 너가 나중에 그 body의 mass properties를 변경한다면, 그러면 그 center of mass는 그 body에서 움직일지도 모르지만, 그 origin position은 변하지 않고, 그 부착된 shapes와 joints는 움직이지 않는다.

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

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

rigid body는 또한 reference의 한 frame이다. 너는 그 frame에서 fixtures와 joints를 정의할 수 있다. 그러한 fixtures와 joint anchors들은 결코 그 body의 local frame에서 움직이지 않는다.

### Damping

Damping은 bodies의 world velocity를 줄이기위해 사용된다. Damping은 friction과 다르다. 왜냐하면 마찰은 오직 contact와만 발생하기 때문이다. Damping은 friction에 대한 대체가 아니고, 그 두 효과는 함께 사용되어질 것이다.

Damping 파라미터들은 0과 무한 사이에 있어야하고, 0은 damping이 없다는 것을 의미하고, 무한은 full damping을 의미한다. 일반적으로 너는 0 ~ 0.1 사이의 damping value를 사용할 것이다. 나는 일반적으로 linear damping을 사용하지 않는데, 왜냐하면 그것은 bodies가 떠다니는 것처럼 보이게 만들기 때문이다.

bodyDef.linearDamping = 0.0f;

bodyDef.angularDamping = 0.01f;

Damping은 안정성과 성능을 위해 근사(approximated)된다. 작은 damping values에서, 그 damping effect는 대개 time step과 독립적이다. 큰 damping values에서, 그 damping effect는 time step과 함께 변할 것이다. 이것은 만약 너가 fixed time step을 사용한다면 문제가 아니다 (추천되는 방법이다).

### Gravity Scale

너는 single body에서 gravity를 조절하기 위해 gravity scale을 사용할 수 있다. 그러나 조심해라. 증가된 gravity는 안정성을 줄일 수 있다.

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

bodyDef.gravityScale = 0.0f;

### Sleep Parameters

sleep은 무엇을 의미하는가? bodies를 시뮬레이션 하는 것은 비싸다. 그래서 우리는 덜 재현(simulate)할 수록 더 좋다. body가 휴식에 들어갈 때, 우리는 그것을 재현하는 것을 멈추고 싶다.

Box2D가 한 body(또는 bodies의 그룹)가 쉬려고하는 것을 결정할 때, 그 body는 CPU overhead가 거의 없는 sleep state에 들어간다. 만약 한 body가 깨어있거나 잠자는 body와 충돌한다면, 그러면 잠자는 body는 깨어난다. Bodies는 또한 만약 그것들에 부착된 joint 또는 contact가 파괴된다면 깨어날 것이다. 너는 또한 수동으로 한 body를 깨울 수 있다.

body definition은 body가 잘 수 있는지와 한 body가 sleeping하도록 생성되었는지를 명시하도록 해준다.

bodyDef.allowSleep = true;

bodyDef.awake = true;

### Fixed Rotation

너는 고정된 회전을 갖는, 캐릭터 같은, rigid body를 원할지도 모른다. 그러한 body는 심지어 무게를 받고있어도 회전해선 안된다. 너는 이것을 얻기위해 fixed rotation setting을 사용할 수 있다:

bodyDef.fixedRotation = true;

fixed rotation flag는 rotational inertia와 그것의 inverse가 0으로 설정되게 한다.

### Bullets

Game simulation usually generates a sequence of images that are played at some frame rate. This is 게임 시뮬레이션은 보통 어떤 프레임율에서 재생되는 이미지들의 한 연속을 생성한다. 이것은 discrete simulation이라고 불려진다. discrete simulation에서, rigid bodies는 한 time step에 큰 양으로 움직일 수 있다. 만약 한 physics engine이 large motion을 다루지 않는다면,너는 어떤 오브젝트들이 부정확하게 서로를 지나가는 것을 볼지도 모른다. 이 효과는 tunneling이라고 불려진다.

기본적으로, Box2D는 dynamic bodies가 static bodies를 지나서 tunneling하는 것을 방지하기 위해 continuous collision detection(CCD)를 사용한다. 이것은 그것들의 이전 position을 새로운 position으로 shapes를 sweep하여 처리된다. 그 engine은 그 sweep동안 새로운 충돌을 찾고, 이러한 충돌에 대한 time of Impact(TOI)를 연산한다. Bodies는 그것들의 첫 번째 TOI로 옮겨지고나서, full time step을 완료하기 위해 sub-step을 수행한다. sub-step 낸에서 추가적인 TOI 이벤트들이 있을지도 모른다.

일반적으로 CCD는 dynamic bodies 사이에 사용되지 않는다. 이것은 성능을 합리적으로 유지하기위해 된다. 어떤 게임 시나리오에서는, 너는 CCD를 사용하는 dynamic bodies가 필요하다. 예를들어, 너는 dynamic bricks의 stack에서 높은 speed를가진 bullet을 쏘길 원할지도 모른다. CCD 없이, 그 bullet은 그 bricks를 tunnel할지도 모른다.

Box2D에서 빠르게 움직이는 오브젝트들은 bullets으로 라벨링 되어질 수 있다. Bullets은 static and dynamic bodies 둘 다에 CCD를 수행할 것이다. 너는 너의 게임 설계를 기반으로 어떤 bodies가 bullets이 되어야 하는지를 결정해야 한다. 만약 너가 한 body가 bullet으로 다뤄져야 한다고 결정한다면, 다음의 설정을 사용해라

bodyDef.bullet = true;

그 bullet flag는 오직 dynamic bodies에 영향을 미친다.

### Activation

너는 한 body가 생성되지만 충돌 또는 dynamics에 참여하지 않기를 원할지도 모른다. 이 상태는 body가 다른 bodies에 의해 깨어나지 않을 것이고, 그 body의 fixtures가 broad-phase에 위치하지 않을 것을 제외하고는 sleeping과 유사하다.

너는 한 body를 inactive state로 생성할 수 있고 나중에 그것을 재활성화 할 수 있다.

bodyDef.active = true;

Joints는 inactive bodies에 연결될지도 모른다. 이러한 joints들은 시뮬레이션되지 않을 것이다. 너는 한 body를 활성화할 때, 그것의 joints들이 비틀어지지 않도록 조심해야만 한다.

한 body를 활성화하는 것은 처음부터 body를 생성하는 것 만큼 비싸다는 것에 주목해라. 그래서 너는 worlds를 흘러가게 하는 것에 있어서 activation을 사용해서 안된다. 메모리를 절약하기 위해 worlds를 흘러가게 하기 위해서 creation/destruction을 사용해라.

### User Data

User data는 void pointer이다. 이것은 너에게 너의 어플리케이션 오브젝트들을 bodies에 연결시킬 고리를 준다. 너는 모든 body user data에 같은 오브젝트 type 사용하도록 일관되어야 한다.

b2BodyDef bodyDef;

bodyDef.userData = &myActor;

## Body Factory

Bodies는 world class에 의해 제공되는 body factor를 사용하여 생성되고 파괴된다. 이것은 world가 효율적인 allocator로 body를 생성하게하고, 그 body를 world data 구조체에 더하도록 한다.

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

... do stuff ...

myWorld->DestroyBody(dynamicBody);

dynamicBody = NULL;

**경고**

너는 결코 body를 생성하기 위해 new 또는 malloc을 사용해서는 안된다. 그 world는 그 body에 대해 모를 것이고, 그 body는 적절히 초기화되지 않을 것이다.

Box2D는 body definition에 대한 참조를 보유하지 않고 또는 그것이 보유하는 데이터의 어떤 것도 유지하지 않는다 (user data pointers를 제외하고). 그래서 너는 임의의 body definitions을 생성할 수 있고, 그 같은 body definitions를 재사용할 수 있다.

Box2D는 너가 너의 b2World object를 삭제하여 bodies를 파괴하는 것을 피하도록 해준다. 그리고 이것은 너를 위해 청소 작업을 한다. 그러나, 너는 너의 게임 엔진에서 유지하는 body pointers들을 null값을 넣는 것에 신경써야 한다.

너가 한 body를 파괴할 때, 그 부착된 fixtures와 joints는 자동적으로 파괴된다. 이것은 너가 shape와 joint pointers를 어떻게 관리할지에 대한 중요한 의미를 갖는다.

## Using a Body

한 body를 생성한 후에, 그 body에 너가 수행할 수 있는 많은 연산들이 있다. 이러한 것은 mass properties를 설정하고, position과 velocity에 접근하고, forces를 적용하고, points와 vectors를 변환하는 것을 포함한다.

### Mass Data

한 Body는 mass (scalar), center of mass (2-vector), 그리고 rotational inertia (scalar)를 가지고 있다. Static bodies에 대해, 그 mass와 rotational inertia는 0으로 설정된다. 한 body가 fixed rotation을 가지고 있을 때, 그것의 rotational inertia는 zero이다.

보통, 한 body의 mass properties는 fixtures가 그 body에 더해질 때 자동적으로 설정된다. 너는 run-time에 한 body의 mass를 또한 조절할 수 있다. 이것은 보통 너가 그 질량을 바꾸는 것을 요구하는 특별한 게임 시나리오를 가질 때 된다.

void SetMassData(const b2MassData\* data);

한 body의 mass를 직접적으로 설정한 후, 너는 fixtures에 의해 관리되는 natural mass로 돌아가길 원할지도 모른다. 너는 다음으로 이것을 할 수 있다 :

void ResetMassData();

그 body의 mass data는 다음의 함수를 통해 이용 가능하다:

float32 GetMass() const;

float32 GetInertia() const;

const b2Vec2& GetLocalCenter() const;

void GetMassData(b2MassData\* data) const;

### State Information

Body의 state에 대한 많은 측면들이 있다. 너는 이 state data를 효율적으로 다음의 함수들을 통해 접근할 수 있다:

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

너는 한 body의 position과 rotation에 접근할 수 있다. 이것은 너의 관련된 game actor를 렌더링할 때 흔하다. 너는 또한 position을 설정할 수 있다. 비록 너가 보통 움직임을 시뮬레이션하기 위해 Box2D를 사용할 것이기 때문에 덜 흔할지라도.

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

const b2Transform& GetTransform() const;

const b2Vec2& GetPosition() const;

float32 GetAngle() const;

너는 local and world coordinates에 있는 mass position의 center에 접근할 수 있다. Box2D에서 내부 시뮬레이션의 대부분은 center of mass를 사용한다. 그러나, 너는 보통 그것에 접근할 필요가 없을 것이다. 대신에 너는 보통 body transform으로 작업할 것이다. 예를들어, 너는 사각형의 body를 가질지도 모른다. 그 body origin은 그 square의 한 모서리이다. 반면에 center of mass는 그 사각형의 중앙에 위치해 있다.

const b2Vec2& GetWorldCenter() const;

const b2Vec2& GetLocalCenter() const;

너는 linear and angular velocity에 접근할 수 있다. 그 linear velocity는 center of mass에 대한 것이다. 그러므로, 그 linear velocity는 mass properties가 변한다면 바뀔지도 모른다.

# Fixtures

## About

Shapes가 bodies를 모르고, physics simulation과 별개로 사용될지도 모른다는 것을 회상해라. 그러므로 Box2D는 shapes를 bodies에 붙일 b2Fixture 클래스를 제공한다. 한 body는 zero 이상의 fixtures를 가질지도 모른다. 여러 fixtures를 가진 한 body는 가끔씩 compound body라고 불려진다.

Fixtures는 다음을 보유한다:

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

이러한 것들은 다음 섹션에서 설명된다.

## Fixture Creation

Fixtures는 fixture definition을 초기화하고 그러고나서 그 definition을 parent body에 전달하여 생성된다.

b2FixtureDef fixtureDef;

fixtureDef.shape = &myShape;

fixtureDef.density = 1.0f;

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

이것은 fixture를 생성하고 그것을 body에 붙인다. 너는 fixture pointer를 저장할 필요가 없다. 그 fixture는 parent body가 파괴될 때 자동적으로 파괴될 것이다. 너는 단일 body에 대해 여러 fixtures를 생성할 수 있다.

너는 parent body에 있는 한 fixture를 파괴할 수 있다. 너는 부숴질 수 있는 한 오브젝트를 모델링하기 위해 이것을 할지도 모른다. 만약 그렇지 않는다면, 그 fixture는 홀로 남고, 그 body destruction은 그 부착된 fixtures들을 파괴하는 것에 신경을 쓰게 할 수 있다.

myBody->DestroyFixture(myFixture);

### Density

Fixture density는 parent body의 mass properties를 연산하기 위해 사용된다. 그 density는 zero 또는 양수일 수 있다. 너는 일반적으로 모든 너의 fixtures에 대해 유사한 densities를 사용해야많 ks다. 이것은 stacking stability를 개선시킬 것이다.

한 body의 질량은 너가 그 density를 설정할 때 조절되지 않는다. 너는 이것이 발생하도록 하기 위해 ResetMassData를 호출해야만 한다.

fixture->SetDensity(5.0f);

body->ResetMassData();

### Friction

마찰은 오브젝트들이 서로를 따라 현실적으로 미끄러지게 만들기 위해 사용된다. Box2D는 static and dynamic friction을 지원하지만, 둘 다에 대해 같은 파라미터를 사용한다. 마찰은 Box2D에서 정확히 재현되고, 그 마찰력은 normal force에 비례한다 (이것은 Coulomb friction이라고 불려진다). 그 friction parameter는 보통 0 ~ 1사이에 설정되지만, 어떤 음의 아닌 값이 될 수 있다. 0의 마찰값은 마찰을 끄고, 1의 값은 마찰을 강하게 만든다. 그 마찰력이 두 shapes간에 연산될 때, Box2D는 두 부모 fixtures의 마찰 파라미터들을 결합한다. 이것은 기하학적 의미로 처리된다:

float32 friction;

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

그래서 만약 한 fixture가 zero friction을 가지고 있다면, 그 contact는 zero friction을 가질 것이다.

너는 b2Contact::SetFrictionb을 사용하여 default 혼합된 friction을 덮어씌울 수 있다. 이것은 보통 b2ContactListener callback에서 처리된다.

### Restitution

Restitution은 오브젝트들이 bounce하도록 만들기 위해 사용된다. Restitution value는 보통 0 ~ 1 사이로 설정된다. 한 공이 테이블 위에서 떨어지는 것을 고려해라. 0의 값은 공이 bounce하지 않을 것이라는 것을 의미한다. 이것은 inelastic collision (비탄성 충돌)이라고 불려진다. 1의 값은 공의 속도는 정확히 반사될 것이라는 것을 의미한다. 이것은 완벽하게 elastic collision이다. Restitution은 다음의 공식을 사용하여 결합된다.

float32 restitution;

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

Restitution은 이 방식으로 결합되는데, 너가 bouncy한 floor를 가지고 있지 않고, bouncy한 super ball를 가질 수 있게 하기 위해서이다.

너는 b2Contact::SetRestitution을 사용하여 default mixed restitution을 덮어씌울 수 있다. 이것은 보통 b2ContactListener callback에서 처리된다.

한 shape가 여러 contacts를 만들 때, restitution은 근사되어 재현된다. 이것은 왜냐하면 Box2D가 iterative (반복적) solver를 사용하고 있기 때문이다. Box2D는 또한 collision velocity가 작을 때 inelastic collisions을 사용한다. 이것은 jitter를 방지하는데 쓰인다. B2Settings.h에서 b2\_velocityThreshold를 보아라.

### Filtering

Collision filtering은 fixtures사이에 충돌을 방지하게 해준다. 예를들어, 너가 한 자전거를 타는 캐릭터를 만든다고 하자. 너는 자전거가 terrain과 충돌하고 그 캐릭터가 terrain과 충돌하기ㅣ를 바라지만, 너는 그 캐릭터가 자전거와 충돌하기를 원하지 않는다 (왜냐하면 그것들이 중첩되어야 하기 때문이다). Box2D는 categories와 groups를 사용하여 그러한 collision filtering을 지원한다.

Box2D는 16개의 충돌 카테고리들을 지원한다. 각 fixture에 대해, 너는 그것이 어떤 카테고리에 속하는지 명시할 수 있다. 너는 또한 이 fixture가 무슨 다른 categories와 충돌할 수 있는지를 명시할 수 있다. 예를 들어, 너는 멀티플레이어 게임에서, 모든 플레이어들은 서로 충돌하지 않고 몬스터들도 서로 충돌하지 않지만, 플레이어들과 몬스터들이 충돌하도록 명시할 수 있다. 이것은 masking bits로 처리된다 예를 들어,:

playerFixtureDef.filter.categoryBits = 0x0002;

monsterFixtureDef.filter.categoryBits = 0x0004;

playerFixtureDef.filter.maskBits = 0x0004;

monsterFixtureDef.filter.maskBits = 0x0002;

여기에 충돌이 발생할 규칙이 있다:

uint16 catA = fixtureA.filter.categoryBits;

uint16 maskA = fixtureA.filter.maskBits;

uint16 catB = fixtureB.filter.categoryBits;

uint16 maskB = fixtureB.filter.maskBits;

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

{

// fixtures can collide

}

Collision groups은 너가 한 내장 group index를 명시하게 한다. 너는 같은 group index를 가진 모든 fixtures들이 항상 충돌하게 하거나 (양수 인덱스) 또는 결코 충돌하지 않게 (음수 인덱스) 할 수 있다. 그룹 인덱스들은 보통 어느정도 관련된 것들에 대해 사용된다. 한 자전의 부품들처럼. 다음의 예제에서, fixture1과 fixture2는 항상 충돌하지만 fixture 3와 fixture4는 결코 충돌하지 않는다.

fixture1Def.filter.groupIndex = 2;

fixture2Def.filter.groupIndex = 2;

fixture3Def.filter.groupIndex = -8;

fixture4Def.filter.groupIndex = -8;

다른 그룹 인덱스들을 가진 fixtures사이의 충돌들은 카테고리와 mask bits에 따라서 필터링된다. 다시 말해서, group filtering은 category filtering보다 더 큰 우선권을 가진다.

부가적인 collision filtering이 Box2D에서 발생하는 것에 주목해라. 여기에 리스트가 있다:

* 한 Static body에 있는 한 fixture는 한 dynamic body와만 충돌할 수 있다.
* Kinematic body에 있는 한 fixture는 dynamic body와만 충돌할 수 있다.
* 같은 body에 있는 Fixtures들은 결코 서로 충돌하지 않는다.
* 너는 선택적으로 한 joint에 의해 연결된 bodies에 있는 fixtures 사이의 충돌을 활성/비활성화 할 수 있다.

가끔씩 너는 한 fixture가 이미 만들어지고 나서 collision filtering을 바꿀 필요가 있을지도 모른다. 너는 b2Fixture::GetFilterData와 b2Fixture::SetFilterData를 사용해서 존재하는 fixture에 b2Filter 구조체를 얻고 설정할 수 있다. Filter data를 바꾸는 것은 다음 time step까지 contacts를 더하거나 제거하지 않을 것이라는 것에 주목해라 (World 클래스 참조).

## Sensors

가끔씩 game logic은 두 fixtures가 충돌하지만 collision response(반응)이 없는 상황을 알 필요가 있다. 이것은 sensors를 사용해서 처리된다. 한 sensor는 충돌을 탐지하지만, response를 만들어내지 않는 fixture이다.

너는 어떤 fixture를 한 sensor가 되도록 지정할 수 있다. Sensors들은 static, kinematic, 또는 dynamic이 될지도 모른다. 너가 body마다 여러 개의 fixtures들을 가질지 모르고, 너가 sensors와 solid fixtures의 혼합해서 가질 수 있다는 것을 기억해라. 또한, sensors는 적어도 한 body가 dynamic일 때 contacts만을 형성한다. 그래서 너는 kinematic vs kinematic, kinematic vs static, static vs static에 대해 contact를 얻지 못할 것이다.

Sensors는 contact points를 생성하지 않는다. 한 sensor의 상태를 얻는 두 가지 방법들이 있다:

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

# Joints

## About

Joints는 bodies를 world에 제약하거나 서로에 제약하는데 사용된다. 게임에서 일반적인 예제들은 ragdolls, 시소, 그리고 도르래를 포함한다. Joints는 흥미로운 움직임을 만들기 위해 많은 다른 방법들로 결합될 수 있다.

몇 가지 joints들은 limits을 제공한다. 그래서 너는 움직임의 범위를 제어할 수 있다. 몇 가지 joint는 motors를 제공하고, 이것은 그 joint가 미리 규정된 force/torque가 초과될 때 까지 미리 규정된 속도로 joint를 움직이게 하는데 사용될 수 있다.

Joint motors는 많은 방식으로 사용될 수 있다. 너는 실제 position과 바라는 position 사이의 차이에 비례하여 joint velocity를 명시하여 positions을 제어하는 motors를 사용할 수 있다. 너는 또한 joint friction을 재현하는 motors를 사용할 수 있다: joint velocity를 0으로 설정하고, 작지만 중요한 최대 motor force/torque를 제공해라. 그러고나서 그 motor는 그 부하가 강해질 때 까지 joint가 움직이는 것을 막으려고 할 것이다.

## The Joint Definition

각 joint type은 b2JointDef로부터 파생되는 definition을 가진다. 모든 joints들은 두 개의 다른 bodies로 연결된다. 한 body는 static일지도 모른다. Static and/or kinematic bodies사이의 joints는 허용되지만, 어떠한 효과도 없고, 연산 시간을 사용한다.

너는 어떤 joint type에 대해 user data를 명시할 수 있고, 부착된 bodies들이 서로 충돌하는 것을 방지하는 flag를 제공할 수 있다. 이것은 실제로 default behavior이다. 그리고 너는 연결된 bodies 사이의 충돌을 허용하기 위해서 collideConnected Bollean을 설정해야만 한다.

많은 joint definitions는 너가 어떤 기하 데이터를 제공할 것을 요구한다. 종종 한 joint는 anchor points에 의해 정의된다. 이러한 것들을 부착된 bodies에 고정된 점들이다. Box2D는 이러한 점들을 local coordinates에서 명시되도록 요구한다. 이 방식으로, 그 joint는 현재 body transforms이 joint constraint를 위반했을 때에도 명시되어질 수 있다 – 한 게임이 저장되고 다시 불러와질 때 흔히 발생하는 것이다. 부가적으로 어떤 joint definitions은 bodies사이의 default relative angle (기본 상대적 각도)를 알 필요가 있다. 이것은 rotation을 정확히 제약하는데 필수적이다.

Geometric data를 초기화하는 것은 지루할 수 있다. 그래서 많은 joints들은 그 작업을 제거할 현재 body transforms을 사용하는 초기화 함수를 가지고 있다. 그러나, 이러한 초기화 함수들은 보통 prototyping을 위해서만 사용되어져야 한다. Production code는 geometry를 직접 정의해야 한다. 이것은 joint behavior를 더 튼튼하게 만들 것이다.

Joint definition data의 나머지는 그 joint type에 의존한다. 우리는 이제 이것들을 다룬다.

## Joint Factory

Joints는 world factory methods를 사용하여 생성되고 파괴된다. 이것은 오래된 문제를 가져온다:

**경고**

New 또는 malloc을 사용하여 스택이나 heap에서 joint를 생성하려고 하지마라. 너는 b2World class의 create와 destroy 메소드를 사용하여 bodies와 joints를 생성하고 파괴할 수 있다.

여기에 revolute joint의 life time에 대한 예시가 있다:

b2RevoluteJointDef jointDef;

jointDef.bodyA = myBodyA;

jointDef.bodyB = myBodyB;

jointDef.anchorPoint = myBodyA->GetCenterPosition();

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

... do stuff ...

myWorld->DestroyJoint(joint);

joint = NULL;

포인터가 파괴되고나서 그것들을 null값을 넣는 것은 항상 좋다. 이것은 만약 너가 그 포인터를 재 사용하려고 한다면 그 프로그램이 통제되는 방식으로 충돌이 일어나도록 할 것이다.

한 joint의 lifetime은 간단하지 않다. 이 경고에 주의를 잘 기울여라:

**경고**

Joints는 한 attached body가 파괴될 때 파괴된다.

이 예방책은 항상 필수적인 것은 아니다. 너는 joints가 항상 attached bodies전에 파괴되도록 게임 엔진을 구성할 지도 모른다. 이 경우에, 너는 listener class를 구현할 필요가 없다. 세부사항을 위해서 Implicit Destruction 섹션을 보아라.

## Using Joints

많은 시뮬레이션들은 joints들을 생성하고, 그것들이 파괴될 때까지 그것들에 다시 접근하지 않는다. 그러나, 풍부한 시뮬레이션을 만들기 위해 너가 사용할 수 있는 joints에 포함된 많은 유용한 데이터가 있다.

첫 번째로, 너는 한 joint로부터 bodies, anchor points, user data를 얻을 수 있따.

b2Body\* GetBodyA();

b2Body\* GetBodyB();

b2Vec2 GetAnchorA();

b2Vec2 GetAnchorB();

void\* GetUserData();

모든 joints들은 reaction force와 torque를 갖는다. 그 reaction force는 anchor point에 있는 body2에 적용된다. 너는 joints를 부수거나 다른 게임 events를 작동시키기 위해 reaction forces를 사용할 수 있다. 이러한 함수들은 어떤 연산을 할지도 모른다. 그래서 만약 너가 그 결과가 필요하지 않다면 그것들을 호출하지 말아라.

b2Vec2 GetReactionForce();

float32 GetReactionTorque();

## Distance Joint

가장 간단한 joint중의 하나는 distance joint이다. 그리고 이것은 두 bodies에 있는 두 점사이의 거리가 상수여야 한다고 말한다. 너가 distance joint를 명시할 때, 두 bodies가 이미 위치해 있어야 한다. 그러고나서 너는 world coordinates에서 두 anchor points를 명시한다. 그 첫 번째 anchor point는 body 1에 연결되고, 그 두 번째 anchor point는 body 2에 연결된다. 이러한 점들은 distance 제약의 길이를 암시한다.



여기에 distance joint definition의 한 예가 있다. 이 경우에, 우리는 그 bodies가 충돌하도록 허용한다.

b2DistanceJointDef jointDef;

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

jointDef.collideConnected = true;

그 distance joint는 또한 spring-damper connection처럼 부드럽게 만들어질 수 있다. 이것이 어떻게 작동하는지 보기위해 testbed에 있는 Web example을 보아라.

부드러움(softness)는 definition에서 두 개의 상수들을 조정하여 얻어진다: frequency와 damping 비율. Frequency가 harmonic oscillator (기타줄 같은)의 frequency라고 생각해라. 그 frequency는 Hertz로 명시된다. 일반적으로 그 frequency는 time step의 frequency의 절반보다 더 작아야 한다. 그래서 만약 너가 60Hz time step을 상요한다면, 그 distance joint의 frequency는 30Hz보다 더 작아야 한다. 그 이유는 Nyquist frequency와 관련이 있다.

그 damping ratio는 차수가 없고, 일반적으로 0 ~ 1사이에 있지만, 더 클 수 있다. 1에서,그 damping은 중요하다 (모든 진동이 사라진다)

jointDef.frequencyHz = 4.0f;

jointDef.dampingRatio = 0.5f;

## Revolute Joint

Revolute joint는 두 bodies가 한 개의 공통 anchor point를 공유하도록 한다. 그 common anchor point는 종종 hinge point라고 불려진다. Revolute joint는 1개의 자유도를 갖는다 : 두 bodies의 relative rotation (상대적 회전). 이것은 joint angle이라고 불려진다.



한 revolute를 명시하기 위해서, 너는 두 개의 bodies와 world space에서 단일 anchor point를 제공할 필요가 있다. 그 초기화 함수는 그 bodies가 이미 정확한 위치에 있다고 가정한다.

이 예제에서, 두 bodies는 첫 번째 body의 center of mass에서 revolute joint에 의해 연결된다.

b2RevoluteJointDef jointDef;

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

그 revolute joint angle은 bodyB가 angle point에 대해 CCW(시계 반대방향)으로 회전할 때 양수이다. Box2D에 있는 모든 각도처럼, 그 revolute angle은 radians로 측정된다. 전통적으로, 그 joint가 Initialize()를 사용하여 생성될 때, 두 bodies의 현재 회전과 상관없이 revolute joint angle은 0이다.

어떤 경우에서, 너는 그 joint angle을 제어하길 원할지도 모른다. 이 것에 대해, 그 revolute joint는 선택적으로 joint limit and/or a motor를 재현할 수 있다. 휘청거릴 것이다.

Joint limit은 그 joint angle이 하한(lower bound)과 상한(upper bound)사이에 남도록 한다. 그 limit은 이것이 발생하도록 필요한 만큼의 torque를 적용할 것이다. 그 limit range는 zero를 포함해야 한다. 만약 그렇지 않다면 그 joint는 시뮬레이션이 시작할 때

한 joint motor는 너가 joint speed를 명시하도록 한다 (각도에 대한 시간의 미분). 그 speed는 음수이거나 양수일 수 있다. 한 motor는 무한의 force를 가질 수 있지만, 이것은 보통 바람직하지 않다. 그 영원한 질문을 상기해라:

"저항할 수 없는 힘이 움직일 수 없는 오브젝트를 만날 때 무엇이 발생하는가?"

나는 너에게 그것이 좋지 않다고 말할 수 있다. 그래서 너는 joint motor에 대해 최대 torque를 제공할 수 있다. 그 joint motor는 만약 요구된 torque가 그 명시된 최대값을 넘지 않는다면 그 명시된 speed를 유지할 것이다. 그 maximum torque가 초과될 때, 그 joint는 느려질 것이고 심지어 반대로 될 수 있다.

너는 joint friction을 재현하기 위해서 joint motor를 사용할 수 있다. Joint speed를 zero로 설정하고, 그 maximum torque를 어느정도 작지만 의미있는 값으로 설정해라. 그 motor는 그 joint가 회전하는 것을 방지할려고 하지만, 의미있는 부하를 만들어 낼 것이다.

여기에 위의 revolute joint definition의 수정이 있다; 이번에, 그 joint는 한 limit과 한 motor를 활성화 했다. 그 motor는 joint friction을 재현하려는 설정이다.

b2RevoluteJointDef jointDef;

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

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

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

jointDef.enableLimit = true;

jointDef.maxMotorTorque = 10.0f;

jointDef.motorSpeed = 0.0f;

jointDef.enableMotor = true;

너는 revolute joint의 angle, speed, 그리고 motor torque에 접근할 수 있다.

float32 GetJointAngle() const;

float32 GetJointSpeed() const;

float32 GetMotorTorque() const;

너는 또한 각 step마다 motor 파라미터들을 업데이트할 수 있다.

void SetMotorSpeed(float32 speed);

void SetMaxMotorTorque(float32 torque);

joint motors들은 몇 가지 흥미로운 능력들을 가지고 있다. 너는 매 step마다 joint speed를 업데이트할 수 있다. 그래서 너는 그 joint가 sine 곡선처럼 또는 너가 원하는 어떤 함수에 따라서 앞뒤로 왔다 갔다 할 수 있게 만들 수 있다.

... Game Loop Begin ...

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

... Game Loop End ...

너는 또한 바라는 joint angle을 추적하기 위해 joint motors를 사용할 수 있다. 예를들어:

... Game Loop Begin ...

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

float32 gain = 0.1f;

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

... Game Loop End ...

일반적으로 너의 gain parameter는 너무 커서는 안 된다. 만약 그렇지 않다면 너의 joint는 불안정해질 지도 모른다.

## Prismatic Joint

Prismatic joint는 명시된 한 축을 따라 두 개의 bodies의 상대적인 translation(평행이동)을 허용한다. Prismatic joint는 relative rotation을 방지한다. 그러므로, prismatic joint는 단일 자유도를 갖는다.



그 prismatic joint definition은 revolute joint 설명과 유사하다; 그냥 각을 translation으로 대체하고 torque를 force로 대체해라. 이 유사점을 사용하는 것은 joint limit과 friction motor를 가진 예제 prismatic joint definition을 제공한다:

b2PrismaticJointDef jointDef;

b2Vec2 worldAxis(1.0f, 0.0f);

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

jointDef.lowerTranslation = -5.0f;

jointDef.upperTranslation = 2.5f;

jointDef.enableLimit = true;

jointDef.maxMotorForce = 1.0f;

jointDef.motorSpeed = 0.0f;

jointDef.enableMotor = true;

그 revolute joint는 스크린 밖으로 나오는 implicit(묵시적) axis를 가지고 있다. 그 prismatic joint는 스크린에 평행한 explicit(명시적) axis가 필요하다. 이 축은 두 bodies에 고정되어있고, 그것들의 움직임을 따른다.

Revolute joint처럼, 그 prismatic joint translation은 Initialize()를 사용해서 joint가 생성될 때 zero 이다. 그래서 너의 lower and upper translation limits사이에 0이 있도록 해라.

Prismatic joint를 사용하는 것은 revolute joint와 비슷하다. 여기에 관련된 멤버 함수들이 있다:

float32 GetJointTranslation() const;

float32 GetJointSpeed() const;

float32 GetMotorForce() const;

void SetMotorSpeed(float32 speed);

void SetMotorForce(float32 force);

## Pulley Joint

Pulley(도르래)는 이상화된(idealized) pulley를 생성하기 위해 사용된다. 그 pulley는 두 bodies를 ground에 연결하고 서로를 연결한다. 한 body가 올라갈 때, 다른 것은 내려간다. 그 pulley rope의 총 길이는 초기 설정에 따라 보존된다.

length1 + length2 == constant

너는 block과 tackle을 시뮬레이션 하는 비율을 제공할 수 있다. 이것은 그 pulley의 한 쪽이 다른 것보다 더 빠르게 확장되도록 한다. 동시에 그 constraint force는 다른 것 보다 한 쪽에서 더 작다. 너는 기계 레버(mechanical leverage)를 생성하는데 이것을 사용할 수 있다.

length1 + ratio \* length2 == constant

예를들어, 만약 그 비율(ratio)이 2라면, 그러면 length 1은 length2의 비율에 두 배로 변할 것이다. 또한 body1에 부착된 rope에서 그 force는 body 2에 부착된 rope의 절반의 constraint force를 가질 것이다.



Pulleys는 한 쪽이 완전히 연장될 때 문제가 될 수 있다. 다른 쪽의 rope는 zero length를 가질 것이다. 이 점에서 그 constraint 방정식은 singular (안 좋음)가 된다. 너는 이것을 방지하기 위해 collision shapes를 설정해야만 한다.

여기에 예제 pulley definition이 있다:

b2Vec2 anchor1 = myBody1->GetWorldCenter();

b2Vec2 anchor2 = myBody2->GetWorldCenter();

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

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

float32 ratio = 1.0f;

b2PulleyJointDef jointDef;

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

Pulley joints는 현재 길이를 제공한다.

float32 GetLengthA() const;

float32 GetLengthB() const;

## Gear Joint

만약 너가 정교한 기계 장치 (mechanical contraption)을 만들고 싶다면, 너는 gears를 사용하고 싶을지도 모른다. 원칙적으로, 너는 gear teeth를 모델링하기 위해 compound shapes를 사용하여 Box2D에서 gears를 생성할 수 있다. 이것은 매우 효율적이지 않고 그걸 쓰는 사람에게는 지루할지도 모른다. 너느 또한 gears를 배치하는데 조심해야만 한다. Teeth mesh가 부드럽게 되어야 해서. Box2D는 gears를 생성하는 더 간단한 method를 가지고 있다 : gear joint.



Gear joint는 revolute and/or prismatic joints에만 연결할 수 있다.

Pulley ratio처럼, 너는 gear ratio를 명시할 수 있다. 그러나, 이 경우에 gear ratio는 음수가 될 수 있다. 또한 한 joint가 revolute joint (angular)이고 다른 joint가 prismatic (translation)일 때, gear ratio는 단위 길이 또는 1 / length를 가질 거라는 것을 명심해라.

coordinate1 + ratio \* coordinate2 == constant

여기에 gear joint 예제가 있다. 그 bodies myBodyA와 myBodyB는 두 개의 joints의 bodies이다. 그것들이 같은 bodies들이 아닌 한.

b2GearJointDef jointDef;

jointDef.bodyA = myBodyA;

jointDef.bodyB = myBodyB;

jointDef.joint1 = myRevoluteJoint;

jointDef.joint2 = myPrismaticJoint;

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

gear joint가 두 개의 다른 joints에 의존하는 것을 주목해라. 이것은 취약한 상황을 만든다. 만약 그러한 joints들이 삭제된다면 무슨 일이 발생하는가?

**경고**

항상 gears에 있는 revolute/prismatic joints 전에 gear joints를 삭제해라. 만약 그렇지 않다면, gear joint에 있는 고아가 되버린 joint pointers 에 의해 나쁜 방식으로 너의 코드는 충돌이 일어날 것이다. 너는 또한 너가 관련된 어떤 bodies를 삭제하기 전에 gear joint를 삭제해야 한다.

## Mouse Joint

Mouse joint는 mouse를 가진 bodies를 조작하기 위해 testbed에서 사용된다. 그것은 cursor의 현재 위치를 향해 한 body에 있는 한 점을 움직이려고 한다. 회전에 대한 제한은 없다.

그 mouse joint definition은 target point, maximum force, frequency, 그리고 damping ratio를 갖는다. 그 target point는 초기에 그 body의 anchor point와 일치한다. 그 maximum force는 여러 dynamic bodies가 상호작용할 때 난폭한 반응들을 방지하기 위해 사용된다. 너는 너가 원하는 만큼 이것을 크게 만들 수 있다. Frequency와 damping ratio는 distance joint와 유사한 spring/damper effect를 만들기 위해 사용될 수 있다.

많은 사용자들은 게임 플레이를 위해 mouse joint를 적용하려고 노력해왔다. 사용자들은 종종 정확한 positioning과 즉각적인 반응을 얻길 원한다. 그 mouse joint는 그 상황에서 매우 잘 작동하지 않는다. 너는 대신에 kinematic bodies를 사용하는 것을 고려하는 것을 원할지도 모른다.

## Wheel Joint

Wheel joint는 bodyA에 있는 선에 bodyB의 한 점을 제한한다. 그 wheel joint는 또한 suspension spring을 제공한다. B2WheelJoint.h와 Car.h를 세부설명을 위해 보아라.



## Weld Joint

Weld joint는 두 bodies 사이에 모든 relative motion을 제한하려고 한다. 그 weld joint가 어떻게 작동하는지 보기 위해 testbed에 있는 Cantilever.h를 보아라.

그것은 부숴질 수 있는 구조물을 정의하기 위해 weld joint를 사용하는 것은 유혹적이다. 그러나, Box2D solver는 iterative이고, 그래서 그 joints는 조금 부드럽다. 그래서 weld joints로 연결된 bodies의 chains은 구부려질 것이다.

대신에 여러 fixtures가 있는 single body로 시작하여 부숴질 수 있는 bodies를 생성하는 것이 더 좋다. 그 body가 부숴질 때, 너는 한 fixture를 파괴하고, 그것을 한 새로운 body에 재생성할 수 있다. Testbed에 Breakable example을 보아라.

## Rope Joint

Rope joint는 두 점 사이의 maximum distance를 제한한다. 이것은 bodies의 chains이 펼쳐지게 하는 것을 방지하는데 유용하다. 심지어 높은 부하 아래에서도. 세부설명을 위해 b2RopeJoint.h와 RopeJoint.h를 보아라.

## Friction Joint

Friction joint는 top-down friction에 사용된다. 그 joint는 2D translational friction과 angular friction을 제공한다. 세부설명을 위해 b2FrictionJoint.h와 ApplyForce.h를 보아라.

## Motor Joint

Motor joint는 target position과 rotation offsets을 명시하여 한 body의 움직임을 통제하게 해준다. 너는 target position과 rotation에 도달하기 위해 적용될 maximum motor force와 torque를 설정할 수 있다. 만약 그 body가 막아진다면, 그것은 멈출 것이고, 그 contact forces는 그 maximum motor force와 torque에 비례할 것이다. B2MotorJoint와 MotorJoint.h를 세부설명을 위해 보아라.

# Contacts

## About

Contact는 두 fixtures 사이의 충돌을 관리하기 위해 Box2D에 의해 생성된다. 만약 그 fixture가 chain shape같은 children을 가지고 있다면, 그러면 한 contact는 각 관련된 child에 대해 존재한다. fixtures들의 다른 종류 사이의 contact를 관리하기 위해 b2Contact로부터 파생된 다른 종류의 contacts들이 있다. 예를들어, polygon-polygon 충돌을 관리하기 위한 contact class가 있고, circle-circle class를 다루기 위한 또 다른 contact class가 있다.

여기에 contacts와 관련된 몇 가지 용어들이 있다.

### contact point

contact point는 두 개의 shapes가 닿는 point이다. Box2D는 작은 개수의 points들을 가진 contact를 근사한다.

### contact normal

contact normal은 한 shape에서 다른 것을 가리키는 단위 벡터이다. 전통적으로 normal은 fixtureA에서 fixtureB를 가리킨다.

### contact separation

Separation(분리)은 penetration(관통)의 반대이다. Separation은 shapes가 중첩될 때 음수이다. Box2D의 미래 버전은 양수 separation을 가진 contact points를 생성하는 것이 가능하다. 그래서 너는 contact points가 보고될 때 그 기호를 체크하길 원할지도 모른다.

### contact manifold

두 개의 convex polygons사이의 Contact는 2개 까지의 contact points를 생성할지도 모른다. 이러한 점들 둘 다 같은 normal을 사용한다. 그래서 그것들은 한 개의 contact manifold로 그룹지어 진다. 그리고 이것은 contact의 연속적인 영역의 근사이다.

### normal impulse

normal force는 shapes가 관통하는 것을 방지하기 위해 contact point에 적용되는 force이다. 편의성을 위해, Box2D는 impulses와 작업한다. 그 normal impulse는 time step으로 곱해진 normal force이다.

### tangent impulse

tangent force는 마찰을 재현하기 위해 contact point에 생성된다. 편의성을 위해 이것은 impulse로 저장된다.

### contact ids

Box2D는 한 time step의 contact force results를 다음 time step의 초기 추측으로 재사용하려고 한다. Box2D는 time steps을 가로질러서 contact points를 매칭시키기 위해 contact ids를 사용한다. 그 ids는 한 contact point를 다른 것과 구분하는데 도와주는 geometric features indices를 포함한다.

Contacts는 두 개의 fixture의 AABB가 중첩될 때 생성된다. 가끔씩 collision filtering은 contacts의 생성을 방지할 것이다. Contacts는 AABB가 중첩되기를 중단하면 파괴된다.

그래서 너는 서로 닿고 있지 않은 fixtures들에 대해 생성된 contacts들이 있다고 추측할지도 모른다. 음, 이것은 옳다. 그것은 “chicken or egg” 문제이다. 우리는 collision을 분석하기 위해 contact가 생성될 때 까지 contact object가 필요한지를 모른다. 우리는 만약 그 shapes가 닿고있지 않다면 바로 그 contact를 삭제할 수 있거나, 또는 우리는 AABB가 overlapping을 중단할 때 까지 기다릴 수 있다. Box2D는 후자의 접근법을 취한다. 왜냐하면 그것은 성능을 향상시키기 위해 system이 정보를 캐싱할 수 있게 하기 때문이다.

## Contact Class

이전에 언급했듯이, contact class는 Box2D에 의해 생성되고 파괴된다. Contact objects는 사용자에 의해 생성되지 않는다. 그러나, 너는 contact class에 접근하고 그것과 상호작용할 수 있다.

너는 raw contact manifold에 접근할 수 있다:

b2Manifold\* GetManifold();

const b2Manifold\* GetManifold() const;

너는 잠재적으로 manifold를 수정할 수 있지만, 이것은 일반적으로 지원되지 않고 고급 사용을 위한 것이다. b2WorldManifo를 얻는 helper function이 있다:

void GetWorldManifold(b2WorldManifold\* worldManifold) const;

이것은 contact points의 world positions을 연산하기 위해 bodies의 현재 positions을 사용한다.

Sensors는 manifolds를 생성하지 않고, 그래서 그것들에 대해 이것을 사용해라:

bool touching = sensorContact->IsTouching();

이 함수는 non-sensors에 대해서 또한 작동한다.

너는 한 contact로부터 그 fixtures들을 얻을 수 있다. 그것 들로부터, 너는 bodies를 얻을 수 있다.

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

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

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

너는 한 contact를 비활성화 할 수 있다. 이것은 b2ContactListener::PreSolve event안에서만 작동한다. 아래에서 이야기 된다.

## Accessing Contacts

너는 몇 가지 방법들로 contacts에 접근할 수 있다. 너는 world에서 직접적으로 contacts와 body 구조체에 접근할 수 있다. 너는 또한 contact listener를 구현할 수 있다.

너는 world에 있는 모든 contacts에 대해 반복한다:

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

{

// process c

}

너는 또한 한 body에 있는 모든 contacts에 대해 반복할 수 있다. 이러한 것은 contact edge 구조체를 사용하여 graph에 저장되어 있다.

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

{

b2Contact\* c = ce->contact;

// process c

}

너는 또한 아래에서 설명되는 contact listener를 사용하여 contacts에 접근할 수 있다.

**경고**

b2World와 b2Body 밖에서 contacts를 접근하는 것은 time step의 중간에 발생하는 몇 가지 순간적인 contacts를 놓칠지도 모른다. 가장 정확한 결과들을 얻기 위해 b2ContactListener를 사용해라.

## Contact Listener

너는 b2ContactListener를 구현하여 contact data를 받을 수 있다. 그 contact listener는 몇 가지 events를 지원한다: 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 \*/ }

};

**경고**

b2ContactListener에 보내지는 포인터들에 대한 참조를 유지하지 말라. 대신에 contact point data를 너의 buffer에 deep copy해라. 아래의 예제는 이것을 하는 한 방법을 보여준다.

Run-time에서, 너는 listener의 한 instance를 생성할 수 있고, 그것을 b2World::SetContactListener로 등록할 수 있다. 너의 listener가 world object가 존재하는 동안의 범위에 남아있도록 해라.

### Begin Contact Event

이것은 두 fixtures가 중첩되기 시작할 때 호출된다. 이것은 sensors와 non-sensors를 위해 호출된다. 이 event는 오직 time step 내에서만 발생할 수 있다.

### End Contact Event

두 개의 fixtures가 중첩하기를 중단할 때 호출된다. 이것은 sensors와 non-sensors를 위해 호출된다. 이것은 한 body가 파괴될 때 호출될지도 모른다. 그래서 이 이벤트는 time step 밖에서 발생할 수 있다.

### Pre-Solve Event

이것은 Collision Detection 후지만 collision resolution전에 호출된다. 이것은 너에게 현재 설정을 기반으로한 contact 를 비활성화할 기회를 준다. 예를들어, 너는 이 콜백을 사용하고 b2Contact::SetEnabled(false)를 사용하여 one-sided platform을 구현할 수 있다. 그 contact는 collision processing을 통해 매번 재활성화 될 것이다. 그래서 너는 매 time-step마다 그 contact을 비활성화 할 필요가 있을 것이다. 그 pre-solve event는 continuous collision detection에 의해 contact마다 time step당 여러 번 실행될지도 모른다.

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

{

b2WorldManifold worldManifold;

contact->GetWorldManifold(&worldManifold);

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

{

contact->SetEnabled(false);

}

}

그 pre-solve event는 또한 point state를 결정하고, collisions의 접근 velocity를 결정하는 좋은 장소이다.

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

Post solve event는 너가 collision impulse 결과들을 모을 수 있는 곳이다. 만약 너가 그 impulses를 신경쓰지 않는다면, 너는 아마도 그 pre-solve event를 구현할 것이다.

Contact callback 내에서 physics world를 변경하는 게임 로직을 구현하는 것은 유혹적이다. 예를들어, 너는 damage를 적용하는 충돌을 가질지도 모르고, 그 관련된 actor와 그것의 rigid body를 파괴할려고 할지도 모른다. 그러나, Box2D는 callback 안에서 physics world를 변경하는 것을 허용하지 않는다. 왜냐하면 너는 Box2D가 현재 처리하고 있는 오브젝트들을 파괴할지도 모른다. 그리고 이것은 고아가 된 pointers들을 만들어 낸다.

Contact points를 처리하는 추천되는 실례는 너가 신경쓰는 모든 contact data를 저장하고, 그 time step후에 그것을 처리하는 것이다. 너는 항상 time step 직후에 contact points를 처리해야만 한다; 만약 그렇지 않는다면 몇 가지 다른 클라이언트 코드는 그 physics world를 변경할지도 모른다. 그리고 이것은 contact buffer를 무효화시킨다. 너가 contact buffer를 처리할 때, 너는 physics world를 변경할 수 있지만, 너는 여전히 너가 contact point buffer에 저장된 pointers들을 고아로 만들지 않도록 신경 쓸 필요가 있다. 그 testbed는 고아가 된 포인터로부터 안전하 예제 contact point processing을 가지고 있다.

이 CollisionProcessing test의 코드는 contact buffer를 처리할 때 어떻게 고아가 된 bodies를 처리하는지를 보여준다. 여기에 발췌문이 있다. Listing에 있는 주석을 읽도록 해라. 이 코드는 모든 contact points가 b2Contactpoint 배열의 m\_points에서 저장되었다고 가정한다.

// 우리는 contact points에 따라 어떤 bodies들을 파괴할 것이다

// 우리는 파괴될 bodies를 저장해야만 한다.

// 왜냐하면 그것들은 여러 contact points에 속할 지도 모르기 때문이다.

const int32 k\_maxNuke = 6;

b2Body\* nuke[k\_maxNuke];

int32 nukeCount = 0;

// contact buffer를 탐색해라.

// 더 무거운 bodies에 닿고있는 bodies를 파괴해라.

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

{

ContactPoint\* point = m\_points + i;

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

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

float32 massA = bodyA->GetMass();

float32 massB = bodyB->GetMass();

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

{

if (massB > massA)

{

nuke[nukeCount++] = bodyA;

}

else

{

nuke[nukeCount++] = bodyB;

}

if (nukeCount == k\_maxNuke)

{

break;

}

}

}

// Sort the nuke array to group duplicates.

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

// Destroy the bodies, skipping duplicates.

int32 i = 0;

while (i < nukeCount)

{

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

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

{

++i;

}

m\_world->DestroyBody(b);

}

## Contact Filtering

종종 게임에서, 너는 모든 오브젝트들이 충돌하기를 원하지 않는다. 예를들어, 오직 어떤 캐릭터들만이 지나갈 수 있는 한 문을 생성하기를 원할지도 모른다. 이것은 contact filtering이라고 불려진다. 왜냐하면 몇 가지 상호작용들이 필터링되기 떄문이다.

Box2D는 b2ContactFilter class를 구현하여 custom contact filtering을 성취하도록 해준다. 이 클래스는 너가 두 개의 b2Shape pointers들을 받는 ShouldCollide function을 구현하는 것을 요구한다. 너의 함수는 만약 그 shapes들이 충돌한다면 true를 반환한다.

ShouldCollide의 기본 구현은 챕터 6 Fixtures에 정의된 b2FilterData를 사용한다.

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

{

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

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

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

{

return filterA.groupIndex > 0;

}

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

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

return collide;

}

Run-time에 너는 너의 contact filter의 instance를 생성하고, 그것을 b2World::SetContactFilter로 등록할 수 있다. 너의 filter가 world가 존재하는 scope내에 머무르도록 해라.

MyContactFilter filter;

world->SetContactFilter(&filter);

// filter remains in scope …

# World Class

### About

b2World class는 bodies와 joints를 포함한다. 그것은 시뮬레이션의 모든 측면들을 관리하고, 비동기 쿼리들을 허용한다 (AABB 쿼리와 ray-casts). Box2D와의 너의 상호작용 대부분은 b2World object가 될 것이다.

### Creating and Destroying a World

World를 생성하는 것은 꽤 간단하다. 너는 gravity vector를 제공할 필요가 있고, bodies가 sleep할 수 있는지를 가리키는 Boolean을 제공할 필요가 있다. 보통 너는 new와 delete를 사용해서 world를 생성하고 파괴할 것이다.

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

... do stuff ...

delete myWorld;

### Using a World

그 world class는 bodies와 joints를 생성하고 파괴하는 factories를 포함한다. 이러한 factories는 bodies와 joints의 섹션에서 나중에 이야기 된다. 내가 지금 여기에서 다룰 b2World와의 몇 가지 다른 상호작용들이 있다.

### Simulation

World class는 시뮬레이션을 주도하기 위해 사용된다. 너는 time step, velocity/position iteration count를 명시한다. 예를들어:

float32 timeStep = 1.0f / 60.f;

int32 velocityIterations = 10;

int32 positionIterations = 8;

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

그 time step후에, 너는 너의 bodies와 joints들의 정보를 검사할 수 있다. 대부분 너는 bodies의 position을 취할 것인데, 너가 너의 actors를 업데이트하고 그것들을 렌더링하기 위해서이다. 너는 너의 game loop 어디에서나 그 time step을 수행할 수 있다. 그러나 너는 상황의 순서를 인짛야 한다. 예를 들어, 만약 그 프레임에 새로운 bodies에 대한 충돌 결과를 얻고자 한다면 너는 time step 전에 bodies를 생성해야만 한다.

내가 위에서 HelloWorld 튜토리얼에서 이야기 했듯이, 너는 fixed time step을 사용해야 한다. 더 큰 time step을 사용하여, 너는 낮은 frame rate scenarios에서 성능을 향상 시킬 수 있다. 그러나 일반적으로 너는 1/30 초보다 더 큰 time step을 사용해선 안된다. 1/60 초의 tie step은 보통 높은 퀄리티의 시뮬레이션을 전달할 것이다.

그 iteration count는 그 constraint solver가 world에 있는 모든 contacts와 joints에 대해 얼마나 많이 탐색하는지를 제어한다. 좀 더 많은 iteration은 더 좋은 시뮬레이션을 만들어 낸다. 그러나 큰 iteration count에 대해 작은 time step을 교환하지 말아라. 60Hz와 10 iterations은 30Hz와 20 iterations보다 훨씬 더 좋다.

Stepping 후에, 너는 너가 bodies에 적용한 어떤 forces든 없애야 한다. 이것은 b2World::ClearForces 명령어로 된다. 이것은 너가 같은 force field를 가진 여러 sub-steps을 가지도록 한다.

myWorld->ClearForces();

### Exploring the World

그 world는 bodies, contacts, 그리고 joints에 대한 container이다. 너는 world에서 body, contact, joint lists들을 가져올 수 있고, 그것들에 대해 반복할 수 있다. 예를 들어 이 코드는 world에 있는 모든 bodies들을 깨운다:

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

{

b->SetAwake(true);

}

불행하게도 진짜 프로그램들은 좀 더 복잡할 수 있다. 예를 들어, 다음의 코드는 망가진다.:

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.

}

}

한 body가 파괴될 때 까지는 모든 것이 괜찮다. 한 body가 파괴된다면, 그것의 다음 포인터는 무효화된다. 그래서 b2Body::GetNext() 호출은 쓰레기 값을 반환할 것이다. 이것에 대한 솔루션은 그 body를 파괴하기전에 다음 포인터를 카피하는 것이다.

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

while (node)

{

b2Body\* b = node;

node = node->GetNext();

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

if (myActor->IsDead())

{

myWorld->DestroyBody(b);

}

}

이것은 안전하게 현재 body를 파괴한다. 그러나, 너는 여러 bodies를 파괴할지도 모르는 한 게임 함수를 호출하고 싶을지도 모른다. 이 경우에 너는 매우 세심할 필요가 있다. 그 솔루션은 어플리케이션 특정하지만,편의성을 위해 나는 그 문제를 해결하는 한 방법을 보여줄 것이다.

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();

}

}

}

명백히 이것이 작동하도록 하기 위해서, GameCrazyBodyDestroy는 그것이 파괴하는 것에 대해 솔직해야 한다.

### AABB Queries

가끔씩 너는 한 영역에 있는 모든 shapes들을 결정하고 싶다. 그 b2World 클래스는 broad-phase data structure를 사용하여 이것에 대해 빠른 log(N) 방법을 가지고 있다. 너는 world coordinates에 있는 AABB와 b2QueryCallback의 구현을 제공한다. 그 world는 너의 클래스를 query AABB와 중첩되는 각 fixture와 호출한다. 그 query를 계속하기위해서 true를 반환하고, 그렇지 않다면 false를 반환해라. 예를들어, 다음의 코드는 명시된 AABB와 잠재적으로 교차하는 모든 fixtures들을 찾고, 모든 관련된 bodies들을 깨운다.

class MyQueryCallback : public b2QueryCallback

{

public:

bool ReportFixture(b2Fixture\* fixture)

{

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

body->SetAwake(true);

// Return true to continue the query.

return true;

}

};

...

MyQueryCallback callback;

b2AABB aabb;

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

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

myWorld->Query(&callback, aabb);

너는 그 콜백들의 순서에 대해 어떤 가정도 할 수 없다.

### Ray Casts

너는 line-of-sight 체크, fire guns, 등을 하기 위해 ray casts를 할 수 있다. 너는 callback class를 구현하고 start와 end points를 제공하여 ray cast를 수행할 수 있다. 그 world class는 그 ray에 의해 hit된 각 fixture와 너의 클래스를 호출한다. 너의 callback의 그 fixture, 교차점, 단위 normal vector 그리고 그 ray를 따라가는 fractional distance 가 제공된다. 너는 그 콜백들의 순서에 대해 어떤 가정을 할 수 없다.

너는 fraction을 반환하여 ray cast를 계속할지를 제어한다. Zero의 fraction을 반환하는 것은 그 ray cast가 끝나야 하는 것을 가리킨다. 1의 fraction은 ray cast 가 마치 어떠한 것과 부딪히지 않은 것처럼 계속 해야하는 것을 가리킨다. 만약 너가 인자 리스트로부터 fraction을 반환한다면, 그 ray는 현재 교차점으로 잘려질 것이다. 그래서 너는 어떤 도형을 ray cast할 수 있고, 모든 도형들, 또는 적절한 fraction을 반환하여 가장 가까운 shape를 ray cast할 수 있다.

너는 또한 그 fixture를 필터링 하기 위해 -1의 fraction을 리턴할지도 모른다. 그러고나서 그 ray cast는 그 fixture가 존재하지 않는 것 처럼 진행할 것이다.

여기에 예제가 있다:

// 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);

**경고**

반올림 에러 때문에, ray casts는 너의 static environment에 있는 폴리곤들 사이의 작은 crack을 통해 들어갈 수 있다. 만약 이것이 너의 프로그램에서 받아들여질 수 없는 것이라면, 너의 폴리곤을 조금 더 크게 해라.

void SetLinearVelocity(const b2Vec2& v);

b2Vec2 GetLinearVelocity() const;

void SetAngularVelocity(float32 omega);

float32 GetAngularVelocity() const;

### Forces and Impulses

너는 한 body에 forces, torques, 그리고 impulses를 적용할 수 있다. 너가 force 또는 impulse를 적용할 때, 너는 그 부하가 적용되는 world point를 제공한다. 이것은 종종 center of mass에 대한 torque를 만들어낸다.

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

void ApplyTorque(float32 torque);

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

void ApplyAngularImpulse(float32 impulse);

force, torque, 또는 impulse를 적용하는 것은 그 body를 깨운다. 가끔씩 이것은 바람직하지 않다. 예를들어, 너는 꾸준한 force를 적용하고있을지 모르고, 그 body가 성능 향상을 위해 sleep하도록 허용하고 싶다. 이 경우에 너는 다음의 코드를 사용할 수 있다.

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

{

myBody->ApplyForce(myForce, myPoint);

}

### Coordinate Transformations

그 body class는 너가 points와 vectors를 local과 world space 사이의 변환을 돕는 몇 가지 utility functions들을 가지고 있다. 만약 너가 이러한 개념을 이해하지 못한다면, Jim Van Verth와 Lars Bishop이 쓴 “Essential Mathematics for Games and Interactive Applications”를 읽어라. 이러한 함수들은 효율적이다 (inlined 처리될 때).

b2Vec2 GetWorldPoint(const b2Vec2& localPoint);

b2Vec2 GetWorldVector(const b2Vec2& localVector);

b2Vec2 GetLocalPoint(const b2Vec2& worldPoint);

b2Vec2 GetLocalVector(const b2Vec2& worldVector);

### Lists

너는 한 body의 fixtures들에 대해 반복할 수 있다. 이것은 만약 너가 그 fixture의 user data에 접근할 필요가 있다면 주로 유용하다.

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

{

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

... do something with data ...

}

너는 유사하게 그 body의 joint list에 대해 반복할 수 있다.

그 body는 또한 연관된 contacts들의 한 리스트를 제공한다. 그 현재 contacts에 대한 정보를 얻기 위해 이것을 사용할 수 있다. 조심해야 한다. 왜냐하면 그 contact list는 이전 time step동안 존재했던 모든 contacts들을 포함하지 않을지도 모른다.

# Loose Ends

## User Data

b2Fixture, b2Body, b2Joint 클래스들은 너가 void pointer로 user data를 붙이게 해준다. 이것은 너가 Box2D 자료구조를 검사하고 너가 너의 게임 엔진에 있는 오브젝트들과 어떻게 연관이 있는지를 결정하고 싶을 때 유용하다.

예를들어, 그 actor에대해 rigid body에 대해 actor pointer를 붙이는 것이 일반적이다. 이것은 circular reference를 설정한다. 만약 너가 actor를 가진다면, 너는 그 body를 얻을 수 있다. 만약 너가 그 body를 가진다면, 너는 그 actor를 얻을 수 있다.

GameActor\* actor = GameCreateActor();

b2BodyDef bodyDef;

bodyDef.userData = actor;

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

여기에 너가 그 user data가 필요한 몇 가지 예제 상황들이 있다:

* 충돌 결과를 사용하여 한 actor에 대한 damage 적용
* 만약 그 player가 axis-aligned box안에 있다면 scripted된 event 실행
* Box2D가 너에게 한 joint가 파괴될 거라고 알려줄 때 게임 구조에 접근하기.

User data가 선택적이고, 너가 그것에 어떤 것이든 넣을 수 있다는 것을 명심해라. 그러나, 너는 일관되어야 한다. 예를 들어, 만약 너가 한 body에 actor pointer를 저장하고 싶다면, 너는 모든 bodies에 한 actor pointer를 유지해야 한다. 한 body에 한 actor pointer를 저장하고 또 다른 body에 foo pointer를 저장하지 말아라. 한 actor pointer를 foo pointer로 캐스팅하는 것은 충돌을 일으킬 것이다.

User data pointers들은 기본적으로 NULL이다.

Fixtures에 대해, 너는 너가 material type, effects hooks, sound hooks,등 과 같은 game에 특정한 정보를 저장하게 해주는 user data structure를 정의하는 것을 고려할지도 모른다.

struct FixtureUserData

{

int materialIndex;

…

};

FixtureUserData myData = new FixtureUserData;

myData->materialIndex = 2;

b2FixtureDef fixtureDef;

fixtureDef.shape = &someShape;

fixtureDef.userData = myData;

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

…

delete fixture->GetUserData();

fixture->SetUserData(NULL);

body->DestroyFixture(fixture);

## Implicit Destruction

Box2D는 reference counting을 사용하지 않는다. 그래서 만약 너가 한 body를 파괴한다면, 그것은 실제로 없어진다. 한 파괴된 body에 대한 포인터에 접근하는 것은 정의되지 않은 행동을 가진다. 다시 말해서, 너의 프로그램이 충돌해서 불 탈 것이다. 이러한 문제들을 고치기 위해서, 그 debug build memory manager는 파괴된 entities에 FDFDFDFD로 채운다. 이것은 어떤 경우에 좀 더 쉽게 문제들을 찾는 것을 도와줄 수 있다.

만약 너가 Box2D entity를 파괴한다면, 너가 그 파괴된 오브젝트에 대한 모든 참조를 제거하도록 하는 것은 너에게 달려있다. 만약 너가 그 entity에 대해 단일 참조만을 가지고 있다면 이것은 쉽다. 만약 너가 여러 참조를 가지고 있다면, 너는 그 raw pointer를 wrap할 handle class를 구현할 것을 고려할지도 모른다.

종종 Box2D를 사용할 때, 너는 많은 bodies, shapes, joints들을 생성하고 파괴할 것이다. 이러한 entities를 관리하는 것은 어느정도 Box2D에 의해 자동화된다. 만약 너가 한 body를 파괴한다면, 그러면 모든 관련된 shapes와 joints들은 자동적으로 파괴된다. 이것은 implicit destruction이라고 불려진다.

너가 한 body를 파괴할 때, 모든 그것의 부착된 shapes, joints, 그리고 contacts들은 파괴된다. 이것은 implicit destruction이라고 불려진다. 그러한 joints들 contacts중 하나에 연결된 어떤 body든 깨어난다. 이 프로세스는 보통 편리하다. 그러나, 너는 한 가지 중요한 문제를 인지해야 한다:

**경고**

한 body가 파괴될 때, 그 body에 부착된 모든 fixtures들과 joints들은 자동으로 파괴된다. 너는 그러한 shapes와 joints에 대해 너가 가지고 있는 어떤 포인터들이든 null 값을 넣어야 한다. 만약 그렇지 않다면, 너가 그러한 shapes 또는 joints에 나중에 접근하거나 파괴하려 한다면, 너의 프로그램은 끔찍히 죽을 것이다.

너가 너의 joint pointers들을 null값을 넣는 것을 돕기위해, Box2D는 너가 구현할 수 있고, 너의 world object에 제공할 수 있는 b2DestructionListener라고 이름지어진 한 listener class를 제공한다. 그러고나서 그 world object는 너에게 한 joint가 implicitly하게 파괴될 때 너에게 알려줄 것이다.

한 joint 또는 fixture가 explicitly하게 파괴될 때 어떠한 알림도 없다는 것에 주목해라. 이경우에 소유권은 명백하고, 너는 그 자리에서 필수적인 cleanup 을 수행할 수 있다. 만약 원한다면, 너는 cleanup code를 중앙화하기 위해 b2DestructionListener의 너만의 구현을 호출할 수 있다.

Implicit destruction은 많은 경우에 훌륭한 편리성을 가진다. 그것은 너의 프로그램을 분리하게 만들 수 있다. 너는 너의 코드 어딘가에서 shapes와 joints에 대한 포인터들을 저장할지도 모른다. 이러한 포인터들은 한 관련된 body가 파괴될 때 고아가 된다. 그 상황은 너가 joints가 관련된 body의 관리와 무관하게 그 코드의 일부분으로 생성될 때를 고려할 때 악화된다. 예를들어, 그 testbed는 스크린에서 bodies의 상호자굥ㅇ하는 조작을 위해 b2MouseJoint를 생성한다.

Box2D는 implicit destruction이 발생할 때 너의 프로그램에 알려주는 callback 메커니즘을 제공한다. 이것은 너의 프로그램에 고아가 된 포인터들을 null값을 넣을 기회를 준다. 이 callback 메커니즘은 이 매뉴얼에서 나중에 설명된다.

너는 b2World가 너에게 한 관련된 body가 파괴 되었었기 때문에 shape 또는 joint가 implicitly하게 파괴될 때 알리도록 해주는 b2DestructionListener를 구현할 수 있다. 이것은 너의 코드가 고아가된 포인터들에 접근하는 것을 방지하는데 도와줄 것이다.

class MyDestructionListener : public b2DestructionListener

{

void SayGoodbye(b2Joint\* joint)

{

// remove all references to joint.

}

};

너는 그러고나서 너의 world object로 너의 destruction listener의 instance를 등록할 수 있다. 너는 이것은 world initialization동안 해야한다.

myWorld->SetListener(myDestructionListener);

## Pixels and Coordinate Systems

Box2D가 MKS(meters, kilograms, and seconds) 단위를 사용하고 각도를 위해서 radians을 사용한다는 것을 상기해라. 너는 미터로 작업하는 것에 문제를 가질지도 모른다. 왜냐하면 너의 게임은 픽셀의 관점에서 표현되기 때문이다. 이것을 testbed에서 다루기 위해, 나는 전체 게임이 meters로 작동하도록 했고, 그 world를 screen space로 스케일하기 위해 OpenGL viewport transformation을 사용한다.

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

gluOrtho2D(lowerX, upperX, lowerY, upperY);

만약 너의 게임이 픽셀단위로 작업해야 한다면, Box2D의 값을 전달할 때 너는 너의 길이 단위를 픽셀에서 미터로 변환해야만 한다. 마찬가지로, 너는 Box2D로부터 받아진 값을 meters에서 pixels로 변환해야 한다. 이것은 물리 시뮬레이션의 안정성을 향상시킬 것이다.

너는 합리적인 변환 factor를 생각해내야 한다. 나는 너의 캐릭터의 사이즈를 기반으로 이 선택을 하기를 제안한다. 너가 미터당 50 픽셀을 사용한다고 결정했다고 하자 (왜냐하면 너의 캐릭턱가 75픽셀 높이이기 때문에). 그러고나서 너는 이러한 공식을 사용하여 pixels에서 미터로 변환할 수 있다:

xMeters = 0.02f \* xPixels;

yMeters = 0.02f \* yPixels;

역으로:

xPixels = 50.0f \* xMeters;

yPixels = 50.0f \* yMeters;

너는 너의 게임 코드에서 MKS units을 사용하는 것을 고려해야 하고, 너가 렌더링할 때 픽셀로 변환해야 한다. 이것은 너의 게임 로직을 간단하게 할 것이고, 에러 가능성을 줄일 것이다. 왜냐하면 렌더링 변환은 작은 코드의 양으로 고립되어 있을 수 있기 때문이다.

만약 너가 conversion factor를 사용한다면, 너는 어떠한 것도 부서지지 않도록 그것을 globally하게 조절하려고 해야한다. 너는 또한 안정성을 향상시키기 위해 그것을 조절할 수 있다.

# Debug Drawing

너는 physics world의 디테일한 drawing을 얻기위해 b2DebugDraw class를 구현할 수 있다. 여기에 그 이용가능한 entities가 있다:

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



이것은 data를 직접 접근하기 보다는 이러한 physics entities를 그리는 선호되는 방법이다.

그 testbed는 debug draw facility와 contact listener를 사용하여 physics entities를 그린다. 그래서 그것은 debug drawing을 어떻게 구현 할 지와 어떻게 contact points를 그릴지에 대해 우선전인 예제로서 역할을 한다.

# Limitations

Box2D는 rigid body physics를 효율적으로 재현하기 위해 몇 가지 근사를 사용한다. 이것은 몇 가지 한계들을 가져온다.

여기에 현재의 한계들이 있다:

1. 더 가벼운 bodies위에 무거운 bodies를 쌓는 것은 안정적이지 않다. 안정성은 그 mass ratio가 10:1을 넘을 때 떨어진다.
2. Joints로 연결된 bodies의 chains은 더 가벼운 body가 더 무거운 body를 받친다면 확장될지도 모른다. 예를 들어, 한 가벼운 bodies의 chain에 연결된 wrecking ball은 안정적이지 않을 지도 모른다. 안정성은 그 mass ratio가 10:1을 넘어갈 때 하락한다.
3. 일반적으로 shape vs shape 충돌에서 약 0.5cm의 slop이 있다.
4. Continuous collision은 joints를 다루지 않는다. 그래서 너는 빠르게 움직이는 오브젝트에 joint가 펼쳐지는 것을 볼지도 모른다.
5. Box2D는 symplectic Euler Integration scheme을 사용한다. 그것은 발사체의 포물선 움직임을 만들어내지 않고, 오직 first-order 정확성만을 가진다. 그러나 그것은 빠르고 좋은 안정성을 가지고 있다.
6. Box2D는 실시간 성능을 위해 iterative solver를 사용한다. 너는 정확히 rigid collisions 또는 pixel perfect 정확성을 얻지 못할 것이다. 그 반복을 증가시키는 것은 정확성을 향상시킬 것이다.

# 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