Box2D v2.3.0 User Manual

*Copyright © 2007-2013 Erin Catto*

Chapter 1 Introduction 2

Chapter 2 Hello Box2D 7

Chapter 3 Common 13

Chapter 4 Collision Module 15

Chapter 5 Dynamics Module 22

Chapter 6 Fixtures 23

Chapter 7 Bodies 26

Chapter 8 Joints 33

Chapter 9 Contacts 42

Chapter 10 World Class 49

Chapter 11 Loose Ends 55

Chapter 12 Debug Drawing 57

Chapter 13 Limitations 59

Chapter 14 References 60

* **소개**
* **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***

joint limit은 한 joint의 움직임 범위를 제한한다. 예를들어, 인간의 팔꿈치는 오직 어떤 범위의 각도만 허용한다.

* ***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**

Hellow 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:

* 한 body를 position, damping, 등등으로 정의하자.
* body를 만들기 위해서 world object를 사용해라.
* shape, 마찰, 밀도 등으로 fixtures를 정의해라.
* 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 gounrd 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, paremeter tuning, debug drawing options을 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
* 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 알고리즘ㄷ르은 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 만들지도 모른다.

// 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 대한 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**

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**

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.

Note that activating a body is almost as expensive as creating the body from scratch. So you should not use activation for streaming worlds. Use creation/destruction for streaming worlds to save memory.

* **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;

* **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.

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.

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.

* **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**

A body has 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.

* **Fixtures**
* **About**

Recall that shapes don’t know about bodies and may be used independently of the physics simulation. Therefore Box2D provides the b2Fixture class to attach shapes to bodies. A body may have zero or more fixtures. A body with multiple fixtures is sometimes called a *compound body.*

Fixtures hold the following:

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

These are described in the following sections.

* **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 not adjusted when you set the density. You must call ResetMassData for this to occur.

fixture->SetDensity(5.0f);

body->ResetMassData();

* **Friction**

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

float32 friction;

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

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

You can override the default mixed friction using b2Contact::SetFriction. This is usually done in the b2ContactListener callback.

* **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(fixtureA->restitution, fixtureB->restitution);

Restitution is combined this way so that you can have a bouncy super ball without having a bouncy floor.

You can override the default mixed restitution using b2Contact::SetRestitution. This is usually done in the b2ContactListener callback.

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. See b2\_velocityThreshold in b2Settings.h.

* **Filtering**

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

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;

Here is the rule for a collision to occur:

uint16 catA = fixtureA.filter.categoryBits;

uint16 maskA = fixtureA.filter.maskBits;

uint16 catB = fixtureB.filter.categoryBits;

uint16 maskB = fixtureB.filter.maskBits;

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

{

// fixtures can collide

}

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

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 can only collide with a dynamic body.
* A fixture on a kinematic body can only collide with a dynamic body.
* Fixtures on the same body never collide with each other.
* You can optionally enable/disable collision between fixtures on bodies connected by a joint.

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).

* **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, kinematic, or dynamic. Remember that you may have multiple fixtures per body and you can have any mix of sensors and solid fixtures. Also, sensors only form contacts when at least one body is dynamic, so you will not get a contact for kinematic versus kinematic, kinematic versus static, or static versus static.

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

* b2Contact::IsTouching
* b2ContactListener::BeginContact and EndContact
* **Joints**
* **About**

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

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.

* **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.

* **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.bodyA = myBodyA;

jointDef.bodyB = myBodyB;

jointDef.anchorPoint = myBodyA->GetCenterPosition();

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

... do stuff ...

myWorld->DestroyJoint(joint);

joint = NULL;

It is always good to nullify your pointer after they are destroyed. This will make the program crash in a controlled manner if you try to reuse the pointer.

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

**Caution**

Joints are destroyed when an attached body is destroyed.

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

* **Using Joints**

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

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

b2Body\* GetBodyA();

b2Body\* GetBodyB();

b2Vec2 GetAnchorA();

b2Vec2 GetAnchorB();

void\* GetUserData();

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

b2Vec2 GetReactionForce();

float32 GetReactionTorque();

* **Distance Joint**

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



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

b2DistanceJointDef jointDef;

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

jointDef.collideConnected = true;

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

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

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

jointDef.frequencyHz = 4.0f;

jointDef.dampingRatio = 0.5f;

* **Revolute Joint**

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



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

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

b2RevoluteJointDef jointDef;

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

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

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

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

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

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

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

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

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

b2RevoluteJointDef jointDef;

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

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

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

jointDef.enableLimit = true;

jointDef.maxMotorTorque = 10.0f;

jointDef.motorSpeed = 0.0f;

jointDef.enableMotor = true;

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

float32 GetJointAngle() const;

float32 GetJointSpeed() const;

float32 GetMotorTorque() const;

You also update the motor parameters each step.

void SetMotorSpeed(float32 speed);

void SetMaxMotorTorque(float32 torque);

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

... Game Loop Begin ...

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

... Game Loop End ...

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

... Game Loop Begin ...

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

float32 gain = 0.1f;

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

... Game Loop End ...

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

* **Prismatic Joint**

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



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

b2PrismaticJointDef jointDef;

b2Vec2 worldAxis(1.0f, 0.0f);

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

jointDef.lowerTranslation = -5.0f;

jointDef.upperTranslation = 2.5f;

jointDef.enableLimit = true;

jointDef.maxMotorForce = 1.0f;

jointDef.motorSpeed = 0.0f;

jointDef.enableMotor = true;

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

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

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

float32 GetJointTranslation() const;

float32 GetJointSpeed() const;

float32 GetMotorForce() const;

void SetMotorSpeed(float32 speed);

void SetMotorForce(float32 force);

* **Pulley Joint**

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

length1 + length2 == constant

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

length1 + ratio \* length2 == constant

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



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

Here is an example pulley definition:

b2Vec2 anchor1 = myBody1->GetWorldCenter();

b2Vec2 anchor2 = myBody2->GetWorldCenter();

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

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

float32 ratio = 1.0f;

b2PulleyJointDef jointDef;

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

Pulley joints provide the current lengths.

float32 GetLengthA() const;

float32 GetLengthB() const;

* **Gear Joint**

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



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

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

coordinate1 + ratio \* coordinate2 == constant

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

b2GearJointDef jointDef;

jointDef.bodyA = myBodyA;

jointDef.bodyB = myBodyB;

jointDef.joint1 = myRevoluteJoint;

jointDef.joint2 = myPrismaticJoint;

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

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

**Caution**

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

* **Mouse Joint**

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

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

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

* **Wheel Joint**

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



* **Weld Joint**

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

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

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

* **Rope Joint**

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

* **Friction Joint**

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

* **Motor Joint**

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

* **Contacts**
* **About**

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

Here is some terminology associated with contacts.

* **contact point**

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

* **contact normal**

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

* **contact separation**

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

* **contact manifold**

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

* **normal impulse**

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

* **tangent impulse**

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

* **contact ids**

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

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

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

* **Contact Class**

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

You can access the raw contact manifold:

b2Manifold\* GetManifold();

const b2Manifold\* GetManifold() const;

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

There is a helper function to get the b2WorldManifold:

void GetWorldManifold(b2WorldManifold\* worldManifold) const;

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

Sensors do not create manifolds, so for them use:

bool touching = sensorContact->IsTouching();

This function also works for non-sensors.

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

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

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

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

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

* **Accessing Contacts**

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

You can iterate over all contacts in the world:

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

{

// process c

}

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

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

{

b2Contact\* c = ce->contact;

// process c

}

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

**Caution**

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

* **Contact Listener**

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

class MyContactListener : public b2ContactListener

{

public:

void BeginContact(b2Contact\* contact)

{ /\* handle begin event \*/ }

void EndContact(b2Contact\* contact)

{ /\* handle end event \*/ }

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

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

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

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

};

**Caution**

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

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

* **Begin Contact Event**

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

* **End Contact Event**

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

* **Pre-Solve Event**

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

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

{

b2WorldManifold worldManifold;

contact->GetWorldManifold(&worldManifold);

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

{

contact->SetEnabled(false);

}

}

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

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

{

b2WorldManifold worldManifold;

contact->GetWorldManifold(&worldManifold);

b2PointState state1[2], state2[2];

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

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

{

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

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

b2Vec2 point = worldManifold.points[0];

b2Vec2 vA = bodyA->GetLinearVelocityFromWorldPoint(point);

b2Vec2 vB = bodyB->GetLinearVelocityFromWorldPoint(point);

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

if (approachVelocity > 1.0f)

{

MyPlayCollisionSound();

}

}

}

* **Post-Solve Event**

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

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

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

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

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

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

// because they may belong to multiple contact points.

const int32 k\_maxNuke = 6;

b2Body\* nuke[k\_maxNuke];

int32 nukeCount = 0;

// Traverse the contact buffer. Destroy bodies that

// are touching heavier bodies.

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

{

ContactPoint\* point = m\_points + i;

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

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

float32 massA = bodyA->GetMass();

float32 massB = bodyB->GetMass();

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

{

if (massB > massA)

{

nuke[nukeCount++] = bodyA;

}

else

{

nuke[nukeCount++] = bodyB;

}

if (nukeCount == k\_maxNuke)

{

break;

}

}

}

// Sort the nuke array to group duplicates.

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

// Destroy the bodies, skipping duplicates.

int32 i = 0;

while (i < nukeCount)

{

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

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

{

++i;

}

m\_world->DestroyBody(b);

}

* **Contact Filtering**

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

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

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

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

{

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

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

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

{

return filterA.groupIndex > 0;

}

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

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

return collide;

}

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

MyContactFilter filter;

world->SetContactFilter(&filter);

// filter remains in scope …

* **World Class**
* **About**

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

* **Creating and Destroying a World**

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

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

... do stuff ...

delete myWorld;

* **Using a World**

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

* **Simulation**

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

float32 timeStep = 1.0f / 60.f;

int32 velocityIterations = 10;

int32 positionIterations = 8;

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

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

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

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

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

myWorld->ClearForces();

* **Exploring the World**

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

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

{

b->SetAwake(true);

}

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

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

{

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

if (myActor->IsDead())

{

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

}

}

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

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

while (node)

{

b2Body\* b = node;

node = node->GetNext();

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

if (myActor->IsDead())

{

myWorld->DestroyBody(b);

}

}

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

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

while (node)

{

b2Body\* b = node;

node = node->GetNext();

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

if (myActor->IsDead())

{

bool otherBodiesDestroyed = GameCrazyBodyDestroyer(b);

if (otherBodiesDestroyed)

{

node = myWorld->GetBodyList();

}

}

}

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

* **AABB Queries**

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

class MyQueryCallback : public b2QueryCallback

{

public:

bool ReportFixture(b2Fixture\* fixture)

{

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

body->SetAwake(true);

// Return true to continue the query.

return true;

}

};

...

MyQueryCallback callback;

b2AABB aabb;

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

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

myWorld->Query(&callback, aabb);

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

* **Ray Casts**

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

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

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

Here is an example:

// This class captures the closest hit shape.

class MyRayCastCallback : public b2RayCastCallback

{

public:

MyRayCastCallback()

{

m\_fixture = NULL;

}

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

const b2Vec2& normal, float32 fraction)

{

m\_fixture = fixture;

m\_point = point;

m\_normal = normal;

m\_fraction = fraction;

return fraction;

}

b2Fixture\* m\_fixture;

b2Vec2 m\_point;

b2Vec2 m\_normal;

float32 m\_fraction;

};

MyRayCastCallback callback;

b2Vec2 point1(-1.0f, 0.0f);

b2Vec2 point2(3.0f, 1.0f);

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

**Caution**

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

void SetLinearVelocity(const b2Vec2& v);

b2Vec2 GetLinearVelocity() const;

void SetAngularVelocity(float32 omega);

float32 GetAngularVelocity() const;

* **Forces and Impulses**

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

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

void ApplyTorque(float32 torque);

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

void ApplyAngularImpulse(float32 impulse);

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

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

{

myBody->ApplyForce(myForce, myPoint);

}

* **Coordinate Transformations**

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

b2Vec2 GetWorldPoint(const b2Vec2& localPoint);

b2Vec2 GetWorldVector(const b2Vec2& localVector);

b2Vec2 GetLocalPoint(const b2Vec2& worldPoint);

b2Vec2 GetLocalVector(const b2Vec2& worldVector);

* **Lists**

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

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

{

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

... do something with data ...

}

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

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

* **Loose Ends**
* **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 objects 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.

User data pointers are NULL by default.

For fixtures you might consider defining a user data structure that lets you store game specific information, such as material type, effects hooks, sound hooks, etc.

struct FixtureUserData

{

int materialIndex;

…

};

FixtureUserData myData = new FixtureUserData;

myData->materialIndex = 2;

b2FixtureDef fixtureDef;

fixtureDef.shape = &someShape;

fixtureDef.userData = myData;

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

…

delete fixture->GetUserData();

fixture->SetUserData(NULL);

body->DestroyFixture(fixture);

* **Implicit Destruction**

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

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

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

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

**Caution**

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

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

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

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

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

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

class MyDestructionListener : public b2DestructionListener

{

void SayGoodbye(b2Joint\* joint)

{

// remove all references to joint.

}

};

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

myWorld->SetListener(myDestructionListener);

* **Pixels and Coordinate Systems**

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

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

gluOrtho2D(lowerX, upperX, lowerY, upperY);

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

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

xMeters = 0.02f \* xPixels;

yMeters = 0.02f \* yPixels;

In reverse:

xPixels = 50.0f \* xMeters;

yPixels = 50.0f \* yMeters;

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

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

* **Debug Drawing**

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

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



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

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

* **Limitations**

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

Here are the current limitations:

* Stacking heavy bodies on top of much lighter bodies is not stable. Stability degrades as the mass ratio passes 10:1.
* 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.
* There is typically around 0.5cm of slop in shape versus shape collision.
* Continuous collision does not handle joints. So you may see joint stretching on fast moving objects.
* Box2D uses the symplectic Euler integration scheme. It does not reproduce parabolic motion of projectiles and has only first-order accuracy. However it is fast and has good stability.
* Box2D uses an iterative solver to provide real-time performance. You will not get precisely rigid collisions or pixel perfect accuracy. Increasing the iterations will improve accuracy.
* **References**

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

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

Real-Time Collision Detection, Christer Ericson, 2005