Box 2D

Box2D

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

Box2D is an open source C engine for simulating rigid bodies in 2D and, like SDL, is also cross-platform and distributed under the zlib license. It has been widely adopted: Nintendo DS, Wii, and several mobile phones (including Android BlackBerry 10 and iOS) as well as most major operating systems. Box2D is the 2D physics engine adopted by Unity and by other game engines, like: Construct 2, Stencyl, LibGDX, GameMaker: Sudio, among others.

Figure 1: Box2D logo.

Box2D is written in portable C11. Most of the types defined in the engine begin with the b2 prefix. Hopefully this is sufficient to avoid name clashing with your application.

Core Concepts

Box2D works with several fundamental concepts and objects:

- **Rigid body** – often referred on the Box2D documentation as a “body”. A **rigid body** is a chunk of matter that is so strong that the distance between any two bits of matter on the chunk is constant. They are hard like a diamond.
- **Shape** – binds collision geometry to a body and adds material properties such as density, friction, and restitution. A **shape** puts collision geometry into the collision system (broad-phase) so that it can collide with other shapes.
- **Constraint** – physical connection that removes degrees of freedom from bodies. In Box2D, each **rigid body** has 3 degrees of freedom (two translation coordinates and one rotation coordinate). If you take a body and pin it to the wall (like a pendulum) you have constrained

the body to the wall. At this point the body can only rotate about the pin, so the constraint has removed 2 degrees of freedom.

- **Contact constraint** – special constraint designed to prevent penetration of rigid bodies and to simulate friction and restitution. **Contact constraints** are automatically created by Box2D.
- **Joint constraint** – This is a constraint used to hold two or more bodies together. Box2D supports several joint types: revolute, prismatic, distance, and more. Joints may have **limits**, **motors**, and/or **springs**.
- **Joint limit** – restricts the range of motion of a joint. For example, the human elbow only allows a certain range of angles.
- **Joint motor** – drives the motion of the connected bodies according to the joint's degrees of freedom. For example, you can use a motor to drive the rotation of an elbow. Motors have a target speed and a maximum force or torque. The simulation will apply the force or torque required to achieve the desired speed.
- **Joint spring** – has a stiffness and damping. In Box2D **spring** stiffness is expressed in terms or Hertz or cycles per second. This lets you configure how quickly a spring reacts regardless of the body masses. **Joint springs** also have a damping ratio to let you specify how quickly the spring will come to rest.
- **World** – collection of bodies, shapes, joints, and contacts that interact together. Box2D supports the creation of multiple worlds which are completely independent.
- **Solver** –The physics **world** has a **solver** that is used to advance time and to resolve contact and joint constraints. The Box2D solver is a high performance sequential solver that operates in order N time, where N is the number of constraints.
- **Continuous collision** – the **solver** advances **rigid bodies** using discrete time steps, which can lead to tunneling (see Figure 2). Box2D contains specialized algorithms to deal with tunneling. First, the collision algorithms can interpolate the motion of two bodies to find the first time of impact (TOI). Second, speculative collision is used to create contact constraints between bodies before they touch.

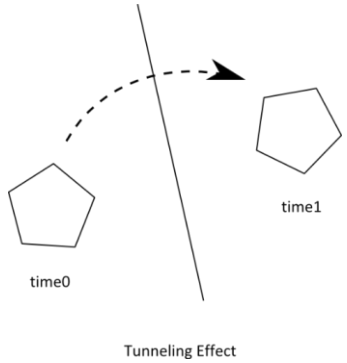

Figure 2 – Tunneling effect. Taken from Box2D manual (https://google.github.io/liquidfun/Programmers-Guide/html/index.html)

- **Events** – World simulation leads to the creation of events that are available at the end of the time step:
    - body movement events;
    - contact begin and end events;
    - contact hit events;

  These events allow your application to react to changes in the simulation.

Modules

Box2D's primary purpose is to provide rigid body simulation. However, there are math and collision features that may be useful apart from the rigid body simulation. These are provided in the `include` directory. Anything in the `include` directory is considered public, while everything in the `src` directory is consider internal.

Units

Box2D works with **floating point** numbers and tolerances have to be used to make Box2D perform well. These tolerances have been tuned to work well with **meters-kilogram-second** (MKS) units. In particular, Box2D has been tuned to work well with moving shapes between 0.1 and 10 meters. So this means objects between soup cans and buses in size should work well. Static shapes may be up to 50 meters long without trouble. If you have a large world, you should split it up into multiple static bodies.

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

Note: since Box2D is tuned for MKS units, you should keep the size of moving objects between 10 centimeters and 10 meters. This way, you will need some scaling system when rendering your environment and actors. **Do not use pixels as size!!!**

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

Another limitation to consider is overall world size. If your world units become larger than 12 kilometers or so, then the lost precision can affect stability.

Note: works best with world sizes less than 12 kilometers. If you are careful with your simulation tuning, this can be pushed up to around 24 kilometers, which is much larger than most game worlds.

Box2D uses **radians** for **angles**. The body **rotation** is stored a **complex number**, so when you access the angle of a body, it will be between $-\pi$ and π radians.

Note: Box2D uses radians, not degrees.

Changing the Length Units

Advanced users may change the length unit by calling `b2SetLengthUnitsPerMeter()` at application startup. If you keep Box2D in a shared library, you will need to call this if the shared library is reloaded.

If you change the length units to pixels you will need to decide how many pixels represent a meter. You will also need to figure out reasonable values for gravity, density, force, and torque. One of the benefits of using MKS units for physics simulation is that you can use real world values to get reasonable results.

It is also harder to get support for using Box2D if you change the unit system, because values are harder to communicate and may become non-intuitive.

Ids and Definitions

Fast memory management plays a central role in the design of the Box2D interface. When you create a world, body, shape or joint, you will receive a handle called an `id`. These ids are opaque and are passed to various functions to access the underlying data.

These ids provide some safety. If you use an id after it has been freed you will usually get an assertion. All ids support 64k generations of safety. All ids also have a corresponding function you can call to check if it is valid.

When you create a world, body, shape, or joint, you need to provide a definition structure. These definitions contain all the information needed to build the Box2D object. By using this approach you can prevent construction errors, keep the number of function parameters small, provide sensible defaults, and reduce the number of accessors.

Here is an example of body creation:

```
b2BodyDef bodyDef = b2DefaultBodyDef();
bodyDef.position = {10.0f, 5.0f};
b2BodyId myBodyId = b2CreateBody(myWorldId, &bodyDef);
```

Notice the body definition is initialized by calling `b2DefaultBodyDef()`. This is needed because C does not have constructors and zero initialization is not suitable for the definitions used in Box2D.

Also notice that the body definition is a temporary object that is fully copied into the internal body data structures. Definitions should usually be created on the stack as temporaries.

This is how a body is destroyed:

```
b2DestroyBody(myBodyId);
myBodyId = b2_nullBodyId;
```

Notice that the body id is set to null using the constant `b2_nullBodyId`. You should treat ids as opaque data, however you may zero initialize all Box2D ids and they will be considered `null`.

Shapes are created in a similar way. For example, here is how a box shape is created:

```
b2ShapeDef shapeDef = b2DefaultShapeDef();
shapeDef.friction = 0.42f;
b2Polygon box = b2MakeBody(0.5f, 0.25f);
b2ShapeId myShapeId = b2CreateCircleShape(myBodyId, &shapeDef, &box);
```

And the shape may be destroyed as follows:

```
b2DestroyShape(myShapeId);
myShapeId = b2_nullShapeId;
```

For convenience, Box2D will destroy all shapes on a body when the body is destroyed. Therefore, you may not need to store the shape id.

Hello Box2D

Along with the distribution of Box2D, there is a “Hello World” unit test written in C. The test creates a large ground box and a small dynamic box. This code does not contain any graphics. All you see is text output in the console of the box’s position over time. However, it is a good example of how to get up and running with Box2D.

Creating a World

Every Box2D program begins with the creation of a world object. The world is the physics hub that manages memory, objects, and simulation. The world is represented by an opaque handle called `b2WorldId`.

It is easy to create a Box2D world. First, create the world definition:

```
// Construct a world object, which will hold and simulate the rigid bodies.
b2WorldDef worldDef = b2DefaultWorldDef();
```

The world definition is a temporary object that you can create on the stack. The function `b2DefaultWorldDef()` populates the world definition with default values. This is necessary because C does not have constructors and zero initialization is not appropriate for `b2WorldDef`.

Now, configure the world gravity vector. Note that Box2D has no concept of up and you may point gravity in any direction you like. Box2D example code uses the positive y-axis as the up direction.

```
/// Define the gravity vector.
worldDef.gravity = {0.0f, -10.0f};
```

Now, create the world object.

```
// Construct a world object, which will hold and simulate the rigid bodies.
b2WorldId worldId = b2CreateWorld(&worldDef)
```

The world creation copies all the data it needs out of the world definition, so the world definition is no longer needed.

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

Creating the Ground Box

Rigid bodies are built using the following steps:

1. Define a **rigid body** with position, damping, etc.
2. Use the world id to create the **rigid body**.
3. Define **shapes** with friction, density, etc.
4. Create **shapes** on the body.

For the first step, we create the ground **rigid body**. For this, you need a **body** definition. With the body definition, we specify the initial position of the ground **rigid body**:

```
// Define the ground body
b2BodyDef groundBodyDef = b2DefaultBodyDef();
groundBodyDef.position = {0, -10};
```

For step 2 the **rigid body definition** and the **world id** are used to create the ground rigid body. Again, the definition is fully copied and may leave scope after the body is created. Bodies are **static by default**. Static bodies don't collide with other static bodies and are immovable.

```
// Call the body factory which allocates memory for the ground body
// from a pool and creates the ground box shape (also from a pool).
// The body is also added to the world.
b2BodyId groundId = b2CreateBody(worldId, &groundBodyDef);
```

Notice that `worldId` is passed by value. Ids are small structures that should be passed by value.

For step 3, create a ground polygon. Use the b2MakeBox() helper function to form the ground polygon into a box shape, with the box centered on the origin of the parent body.

```
// Define the ground box shape. The extents are the half-widths of the box.
b2Polygon groundBox = b2MakeBox(50.0f, 10.0f);
```

The `b2MakeBox()` function takes the **half-width** and **half-height** (extents). So, in this case, the ground box is 100 units wide (**x**-axis) and 20 units tall (**y**-axis). Box2D is tuned for meters, kilograms, and seconds. So, you can consider the extents to be in meters. Box2D generally works best when objects are the size of typical real world objects. For example, a barrel is about 1 meter tall. Due to the limitations of floating point arithmetic, using Box2D to model the movement of glaciers or dust particles is not a good idea.

Let’s finish the ground body in step 4 by creating the shape. For this step you need to create a shape definition which works fine with the default value.

```
// Add the box shape to the ground body.
b2ShapeDef groundShapeDef = b2DefaultShapeDef();
b2CreatePolygonShape(groundId, &groundShapeDef, &groundBox);
```

Box2D does not keep a reference to the shape data. It copies the data into the internal data structures.

Note that every shape must have a parent body, even shapes that are static. You may attach multiple shapes to a single parent body.

When you attach a shape, the shape's coordinates become local to the body. So when the body moves, so does the shape. A shape's world transform is inherited from the parent body. A shape does not have a transform independent of the body. So we don't move a shape around on the body. Moving or modifying a shape that is on a body is possible with certain functions, but it should not be part of normal simulation. The reason is simple: a body with morphing shapes is not a rigid body, but Box2D is a rigid body engine. Many of the algorithms in Box2D are based on the rigid body model. If this is violated you may get unexpected behavior.

Creating a Dynamic Rigid Body

Right now, you have a ground rigid body. You can use the same technique to create a dynamic rigid body. The main difference, besides its dimensions, is that you must establish the dynamic body's **mass properties**.

First, you create the rigid body by using `CreateBody`. By default, rigid bodies are static, so you must set the `b2BodyType` at construction time to make the body dynamic. You should also use the body definition to put the body at the intended position for simulation. Creating a body then moving it afterwards is very inefficient and may cause lag spikes, especially if many bodies are created at the origin.

```
// Define the dynamic body. We set its position and call the body factory.
b2BodyDef bodyDef = b2DefaultBodyDef();
bodyDef.type = b2_dynamicBody;
bdyDef.position = { 0.0f, 4.0f };
b2BodyId bodyId = b2CreateBody(worldId, &bodyDef);
```

Caution: you must set the body type to `b2_dynamicBody` if you want the body to move in response to forces (such as gravity).

Next, we are going to create and attach a polygon shape using a shape definition. First, we create a box shape:

```
// Define another box shape for our dynamic body.
b2Polygon dynamicBox = b2MakeBox(1.0f, 1.0f);
```

Next, we will create a shape definition for the box. Notice that we will set density to 1. The default density is 1, so this is unnecessary. Also, the friction on the shape is set to 0.3

```
// Define the dynamic body shape.
b2ShapeDef shapeDef = b2DefaultShapeDef();
// Set the box density to be non-zero, so it will be dynamic.
shapeDef.density = 1.0f;
// Override the default friction.
shapeDef.friction = 0.3f;
```

Note: A dynamic body should have at least one shape with a non-zero density. Otherwise, you will get strange behavior.

Using the shape definition, you can now create the shape. This automatically updates the mass of the rigid body. You can add as many shapes as you like to a rigid body. Each one will contribute to the total mass.

```
// Add the shape to the body.
b2CreatePolygonShape( bodyId, &shapeDef, &dynamicBox );
```

You now have everything initialized. We are now ready to run the simulation.

Simulating the World

At this step, you have initialized the ground box and a dynamic box. Now, we are ready to set gravity loose to do its thing. But first, there are some details to take into consideration.

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

```
// Prepare for simulation. Typically we use a time step of 1/60 of a
// second (60Hz) and 10 iterations. This provides a high quality simulation
// in most game scenarios.
float32 timeStep = 1.0f / 60.0f;
```

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

Box2D uses sub-stepping as a means of constraint iteration. It lets the simulation move forward in time by small amounts and each constraint gets a chance to react to the changes.

The suggested sub-step count for Box2D is 4. You can tune this number to your liking, just keep in mind that this has a trade-off between performance and accuracy. Using fewer sub-steps increases performance but accuracy suffers. Likewise, using more sub-steps decreases performance but improves the quality of your simulation. For this example, we will use 4 sub-steps.

```
int subStepCount = 4;
```

Note that the time step and the sub-step count are related. As the time-step decreases, the size of the sub-steps also decreases. For example, at 60Hz time step and 4 sub-steps, the sub-steps operate at 240Hz. With 8 sub-steps the sub-step is 480Hz!

We are now ready to begin the simulation loop. In your game the simulation loop can be merged with your game loop. In each pass through your game loop you call `b2World_Step()`. Just one call is usually enough, depending on your frame rate and your physics time step. I recommend this article Fix Your Timestep! to run your game simulation at a fixed rate.

The Hello World test was designed to be simple, so it has no graphical output. The code prints out the position and rotation of the dynamic body. Here is the simulation loop that simulates 90 time steps for a total of 1.5 seconds of simulated time.

```
// This is our little game loop.
for (int32 i = 0; i < 90; ++i)
{
	// Instruct the world to perform a single step of simulation.
	// It is generally best to keep the time step and iterations fixed.
	b2World_Step( worldId, timeStep, subStepCount );

	// Now print the position and angle of the body.
	position = b2Body_GetPosition( bodyId );
	rotation = b2Body_GetRotation( bodyId );

	printf("%4.2f %4.2f %4.2f\n", position.x, position.y, b2Rot_GetAngle(rotation));
}
```

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

```
0.00 4.00 0.00
0.00 3.99 0.00
0.00 3.98 0.00
0.00 3.97 0.00
...
0.00 1.25 0.00
0.00 1.13 0.00
0.00 1.01 0.00
```

Cleanup

When you are done with the simulation, you should destroy the world.

```
b2DestroyWorld(worldId);
```

Collision Detection

Bullets

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

By default, Box2D uses continuous collision detection (CCD) to prevent dynamic bodies from tunneling through static bodies. This is done by sweeping shapes from their old position to their new

positions. The engine looks for new collisions during the sweep and computes the time of impact (TOI) for these collisions. Bodies are moved to their first TOI at the end of the time step.

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

Fast moving objects in Box2D can be configured as bullets. Bullets will perform CCD with all body types, but not other bullets. You should decide what bodies should be bullets based on your game design. If you decide a body should be treated as a bullet, use the following setting:

```
bodyDef.isBullet = true;

// (...)

shape_def.enableSensorEvents = true;
```

The bullet flag only affects dynamic bodies. I recommend using bullets sparingly.

Sensors

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

You can flag any shape as being a sensor. Sensors may be static, kinematic, or dynamic. Remember that you may have multiple shapes per body and you can have any mix of sensors and solid shapes. Also, sensors only form contacts when at least one body is dynamic, so you will not get sensors overlap detection for kinematic versus kinematic, kinematic versus static, or static versus static. Finally sensors do not detect other sensors.

Sensor overlap detection is achieved using events, which are described below.

Sensor Events

Sensor events are available after every call to **`b2World_Step()`**. Sensor events are the best way to get information about sensors overlaps. There are events for when a shape begins to overlap with a sensor.

```
b2World_Step(*world_id, delta_time, sub_step_count);
b2SensorEvents sensorEvents = b2World_GetSensorEvents(*world_id);
for (int i = 0; i < sensorEvents.beginCount; ++i)
{
  b2SensorBeginTouchEvent* beginTouch = sensorEvents.beginEvents + i;
  Actor* colliding_actor = reinterpret_cast<Actor*>(b2Shape_GetUserData(beginTouch->sensorShapeId));

  // process begin event
}
```