Box2D C++ tutorials - Top-down car physics

Last edited: March 19 2014

Halt! You should have a good understanding of
the basic tutorials before venturing further.

Top-down car physics

The discussion of how 'top down' car physics might be implemented in Box2D comes up fairly often, so I thought I would give it a try and make a topic for it. Usually a top-down car is modelled in a zero-gravity world, represented by one body for the chassis and four separate bodies for the wheels. Depending on how realistic a simulation is required it might be good enough to just use one body for the chassis and not worry about having separate wheels.

In either case the crux of the problem is preventing a body from moving in one local axis (tire should not slide sideways) while still allowing it to move freely in the other local axis (tire should be able move back and forwards). This in itself is not such a difficult feat, but the trick is getting it to feel nice for the user when they control the car. If the lateral velocity is simply killed completely the car will feel like it's on rails, and we might actually want to allow the car to skid in some situations, and behave differently on various surfaces etc. Before we get started you might like to take a look at Doug Koellmer's excellent implementation of top-down cars in Flash: qb2DemoReel.swf (click the 'Next demo' button a couple of times). This is the kind of thing we're aiming for.

The basic procedure is to find the current lateral velocity of a body and apply an impulse that will cancel out that velocity. We will start with just one body to represent a tire, and later attach four of these to another body for a more complex simulation. Since all the tires do the same thing we can make a class for them. Here is the starting point, a class which has a b2Body pointer as a member variable and sets it up with a simple box shape.

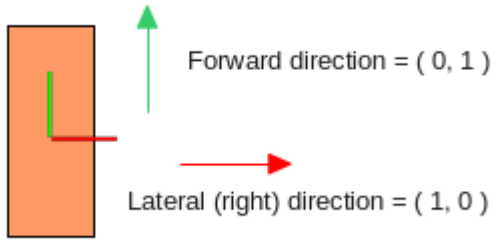
```
1    class TDTire {
2    public:
3        b2Body* m_body;
4
5        TDTire(b2World* world) {
6            b2BodyDef bodyDef;
7            bodyDef.type = b2_dynamicBody;
8            m_body = world->CreateBody(&bodyDef);
9
10           b2PolygonShape polygonShape;
11           polygonShape.SetAsBox( 0.5f, 1.25f );
12           m_body->CreateFixture(&polygonShape, 1);//shape, density
```

```
13
14          m_body->SetUserData( this );
15      }
16
17      ~TDTire() {
18          m_body->GetWorld()->DestroyBody(m_body);
19      }
20  };
```

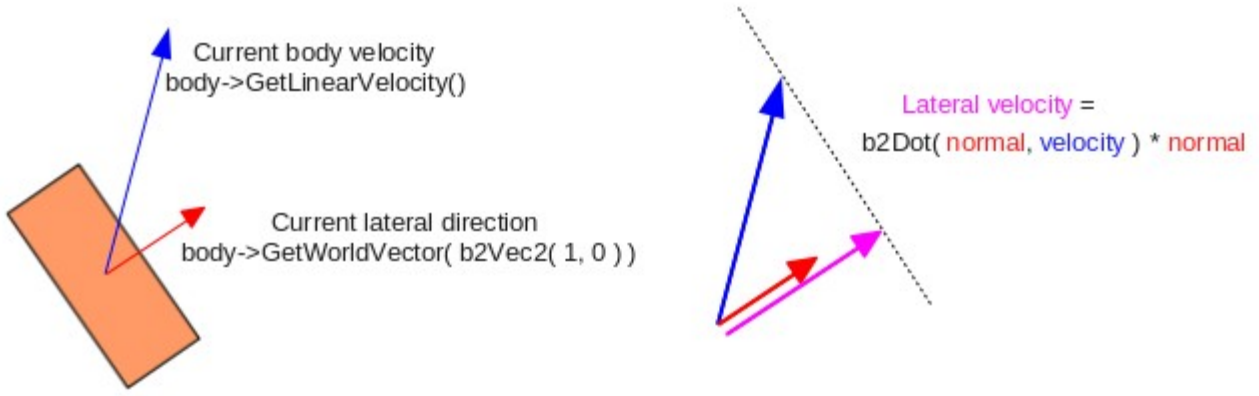
Here I have used the shortcut version of CreateFixture which does not require a fixture def. Note that we've also set the user data of the created b2Body to this class, so the physics body and the game logic class both have a reference to each other.

Killing lateral velocity

To cancel out the lateral velocity we first need to know what it is. We can find it by projecting the current velocity of the body onto the current normal vector for the 'sideways' direction of the tire. Let's say that in the local coordinates of the tire (0,1) will be forwards and (1,0) will be rightwards. We can use GetWorldVector on these to get the current orientation of these directions in world coordinates.



For example suppose the tire is rotated a little, and moving upwards as shown in the next diagram. We want to 'project' the blue vector onto the red one to see how long it would be if it was only going in the red vector's direction.



So we can add a function like this to the tire class:

```
    b2Vec2 getLateralVelocity() {
1       b2Vec2 currentRightNormal = m_body->GetWorldVector( b2Vec2(1,0)
2 );
3       return b2Dot( currentRightNormal, m_body->GetLinearVelocity() )
4 * currentRightNormal;
    }
```

Okay, now to apply an impulse to get rid of that part of the velocity. This is very similar to the final section of the moving at constant speed topic where we had a desired velocity, and we applied an impulse multiplied by the mass of the body to make it reach that velocity in a single time step. Let's add a function to the tire class that we can call every time step to let it do this:

```
1   void updateFriction() {
2       b2Vec2 impulse = m_body->GetMass() * -getLateralVelocity();
3       m_body->ApplyLinearImpulse( impulse, m_body->GetWorldCenter() );
4   }
```

At this stage if you create one of these tires bodies in the world and try pushing it around with the mouse you will notice that the sideways velocity is indeed 'killed' by this impulse. If you push the tire in the forward direction you will find that it tends to go around in a circle, almost like a real tire does as when you roll it and it starts to lose speed.

You will also see that it is still free to rotate around it's center as much as it likes which is a little unrealistic. A real car tire doesn't really do that, so let's kill the angular velocity in a similar way to the lateral velocity. Rotations are a little easier because we don't have to do that vector projection stuff - add this to the updateFriction function:

```
1   m_body->ApplyAngularImpulse( 0.1f * m_body->GetInertia() * -m_body-
>GetAngularVelocity() );
```

The value 0.1 is just something I decided on by playing around with it a bit to get something that looked like what I remember seeing the last time I spun a car tire around :) If we killed the rotation completely (try it) the tire looks like it is on a rail and it can't go anywhere but in a straight line. Another reason not to completely kill the rotation is that pretty soon we will want to let the user turn this body to drive it around.

Finally, you may have noticed that the tire is able to roll on forever in its 'forward' direction, so let's apply a drag force to make it roll to a stop eventually.

```
1   b2Vec2 currentForwardNormal = getForwardVelocity();
2   float currentForwardSpeed = currentForwardNormal.Normalize();
3   float dragForceMagnitude = -2 * currentForwardSpeed;
4   m_body->ApplyForce( dragForceMagnitude * currentForwardNormal,
m_body->GetWorldCenter() );
```

Once again this goes in updateFriction and the value 2 comes from a bit of fiddling and tweaking. Of course all you smart people out there automatically knew that getForwardVelocity() is the same as getLateralVelocity() but works with a local vector of (0,1) instead of (1,0) right?

Controlling a tire

Before we get to making a car with four tires, we need to take care of a few more things that a single tire should do. We will at least need to make it move forward and backwards, and we would also like it to skid realistically and handle different surfaces too. Let's focus on getting all of this working well with one tire, then it will be easy to set up a car with four.

To test situations where the tire is subject to a variety of movements we can first pretend that this single tire itself is a car, and let the user rotate it directly. Here is a basic way of keeping track of which keys (w/a/s/d) the user is currently pressing:

```
1   //global scope
2   enum {
```

```
3       TDC_LEFT   = 0x1,
4       TDC_RIGHT  = 0x2,
5       TDC_UP     = 0x4,
6       TDC_DOWN   = 0x8
7   };
8
9   //testbed Test class variable
10  int m_controlState;
11
12  //testbed Test class constructor
13  m_controlState = 0;
14
15  //testbed Test class functions
16  void Keyboard(unsigned char key) {
17      switch (key) {
18        case 'a' : m_controlState |= TDC_LEFT;  break;
19        case 'd' : m_controlState |= TDC_RIGHT; break;
20        case 'w' : m_controlState |= TDC_UP;    break;
21        case 's' : m_controlState |= TDC_DOWN;  break;
22        default: Test::Keyboard(key);
23      }
24  }
25  void KeyboardUp(unsigned char key) {
26      switch (key) {
27        case 'a' : m_controlState &= ~TDC_LEFT;  break;
28        case 'd' : m_controlState &= ~TDC_RIGHT; break;
29        case 'w' : m_controlState &= ~TDC_UP;    break;
30        case 's' : m_controlState &= ~TDC_DOWN;  break;
31        default: Test::Keyboard(key);
32      }
33  }
```

Note: KeyboardUp is available in the latest Box2D testbed, but if you are using v2.1.2 from the other tutorials you won't find it. You can get the latest version of Box2D, or add the function to the testbed yourself (it's easy, just mimic the Keyboard function :)

Now let's add a function to the tire class to do something clever with that input state:

```
1   //tire class variables
2   float m_maxForwardSpeed;  // 100;
3   float m_maxBackwardSpeed; // -20;
4   float m_maxDriveForce;    // 150;
5
6   //tire class function
7   void updateDrive(int controlState) {
8       //find desired speed
9       float desiredSpeed = 0;
10      switch ( controlState & (TDC_UP|TDC_DOWN) ) {
11          case TDC_UP:   desiredSpeed = m_maxForwardSpeed;  break;
12          case TDC_DOWN: desiredSpeed = m_maxBackwardSpeed; break;
13          default: return;//do nothing
14      }
15
16      //find current speed in forward direction
17      b2Vec2 currentForwardNormal = m_body->GetWorldVector(
18 b2Vec2(0,1) );
19      float currentSpeed = b2Dot( getForwardVelocity(),
20 currentForwardNormal );
21
22      //apply necessary force
23      float force = 0;
24      if ( desiredSpeed > currentSpeed )
```

```
            force = m_maxDriveForce;
        else if ( desiredSpeed < currentSpeed )
            force = -m_maxDriveForce;
        else
            return;
        m_body->ApplyForce( force * currentForwardNormal, m_body-
>GetWorldCenter() );
    }
```

Play around with the speed and force values to get something you like. At the start of the topic I wasn't thinking about dimensions too much and my tire is a rather unrealistic one meter wide, so those speeds are not real-world values either.

Now that the tire can move back and forwards, let's also make it turn by applying some torque when the a/d keys are pressed. Because our end-goal is to attach these tires to a car body, this part of the program will be dropped soon so it's just a crude way to get some turning happening so we can test the next part - skidding and surfaces. On the other hand if you were actually intending to model the car as a single body, you would want to refine this to be more sensible, eg. not letting the car turn unless it is moving etc.

```
void updateTurn(int controlState) {
    float desiredTorque = 0;
    switch ( controlState & (TDC_LEFT|TDC_RIGHT) ) {
        case TDC_LEFT:  desiredTorque = 15;  break;
        case TDC_RIGHT: desiredTorque = -15; break;
        default: ;//nothing
    }
    m_body->ApplyTorque( desiredTorque );
}
```

Allowing skidding

At this point we have a controllable body which behaves very well according to our original plan of killing the lateral velocity. This is all very well if you want to simulate slot-cars which stick to their track like glue, but it feels a bit more natural if the car can skid a bit. Unfortunately this is really really hard... haha just kidding. Actually we have already done it - remember how when we killed the lateral velocity we killed it completely, right? We simply calculated the necessary impulse and applied it, like a boss. That's not very realistic because it means the tire will *never* slip sideways. So all we need to do is restrict that impulse to some maximum value, and the tire will slip when the circumstances require a greater correction than allowable. This is only one extra statement in the updateFriction function:

```
    //in updateFriction, lateral velocity handling section
  b2Vec2 impulse = m_body->GetMass() * -getLateralVelocity();
//existing code
  if ( impulse.Length() > maxLateralImpulse )
      impulse *= maxLateralImpulse / impulse.Length();
  m_body->ApplyLinearImpulse( impulse, m_body->GetWorldCenter() );
 //existing code
```

I found that a value of 3 for the maxLateralImpulse would allow only a very small amount of skidding when turning at high speeds, a value of about 2 gave an effect like a wet road, and a value of 1 reminded me of a speedboat turning on water. These values will need to be adjusted when the wheels are joined to the car chassis anyway, so don't get too fussy with them just yet.

Source code up to this point