Standard Physics Engine:

Usually comprises of the following components:

* A world:
  + It is the data structure that contains all the **objects** in the game world, this has **parameters** such as *gravity*. Some engines allow you to have multiple worlds.
* An Integrator:
  + The **algorithm** that runs each “Tick” or “Step”, it calculates the acceleration, velocity, position of objects in the world.
  + The more “Ticks”, the more accurate the movements will be.
  + Developers have no control over the inner working of this.
* Physics Bodies:
  + These are called rigid bodies, unless fluid(=water); they are things that have a mass, inertia, position and velocity.
  + The **integrator** moves them around based on physics rules.
* Colliders:
  + These are the **shapes** of the objects, they determine how two bodies collide with each other. A body is just an abstract thing that has a mass, **colliders** give them shape and behaviour.
* Constraints:
  + They connect bodies together, permanently or based on some form of logic (elastic, springs, hinges etc)

Steps to follow when using a physics engine:

1. Create the world.
2. Create bodies and attach colliders to them.
3. Each Update() step the physics.
4. Update Entity positions to the one of the physics bodies.

Physics world is separate from the game word, it is left to Box2D to manage.

After a simulation step, we look at the new position of the objects in the physics world and copy the new position into the real/game world render objects.

To have **interactivity**, we need to put some logic into the physics world.

Position and velocity of physics bodies can be manually set, but it isn’t ideal as it breaks the rules of physics that B2D wants to use. (Things don’t teleport in real life, we need to use “**impulses**”.)

**Impulses** are momentary forces that are applied to a body for one frame, like a little push, to move something around. Impulses obey to the rules of physics = heavier objects will need a larger impulse.

Working with Box2D:

Three important factors:

1. It has its own **vector maths classes** that we must convert to/from
2. Its world goes upwards: positive Y goes towards the top of the screen
3. It has a “scale”, we render things in “pixels”
   1. A sf::box is 100 pixels, but how much is it in real measures?
   2. 1 unit = 1 meter when working in 3D games
   3. **B2D has 30 units = 1 pixel**
4. **Creating the world**

(need to add correct include statements)

1. *//main.cpp*
2. b2World**\*** world;
3. **void** **init**() {
4. **const** b2Vec2 gravity(0.0f, **-**10.0f);
5. *// Construct a world, which holds and simulates the physics bodies.*
6. world **=** **new** b2World(gravity);
7. ...
8. }

2. **Creating bodies**

First 3 functions are **conversion helper functions**🡪 they deal with translating between the two worlds (physics and ‘real’).

The CreatePhysicsBox() has all the B2D logic required to add a body to the scene.

The last function is an overload of the fourth, it takes a sf::RectangleShape rather than position+size

*//main.cpp*

*// 1 sfml unit = 30 physics units*

**const** **float** physics\_scale **=** 30.0f;

*// inverse of physics\_scale, useful for calculations*

**const** **float** physics\_scale\_inv **=** 1.0f **/** physics\_scale;

*// Magic numbers for accuracy of physics simulation*

**const** int32 velocityIterations **=** 6;

**const** int32 positionIterations **=** 2;

*//Convert from b2Vec2 to a Vector2f*

**inline** **const** Vector2f **bv2\_to\_sv2**(**const** b2Vec2**&** in) {

**return** Vector2f(in.x **\*** physics\_scale, (in.y **\*** physics\_scale));

}

*//Convert from Vector2f to a b2Vec2*

**inline** **const** b2Vec2 sv2\_to\_bv2(**const** Vector2f**&** in) {

**return** b2Vec2(in.x **\*** physics\_scale\_inv, (in.y **\*** physics\_scale\_inv));

}

*//Convert from screenspace.y to physics.y (as they are the other way around)*

**inline** **const** Vector2f invert\_height(**const** Vector2f**&** in) {

**return** Vector2f(in.x, gameHeight **-** in.y);

}

*//Create a Box2D body with a box fixture*

b2Body**\*** CreatePhysicsBox(b2World**&** World, **const** **bool** dynamic, **const** Vector2f**&** position, **const** Vector2f**&** size) {

b2BodyDef BodyDef;

*//Is Dynamic(moving), or static(Stationary)*

BodyDef.type **=** dynamic **?** b2\_dynamicBody **:** b2\_staticBody;

BodyDef.position **=** sv2\_to\_bv2(position);

*//Create the body*

b2Body**\*** body **=** World.CreateBody(**&**BodyDef);

*//Create the fixture shape*

b2PolygonShape Shape;

Shape.SetAsBox(sv2\_to\_bv2(size).x **\*** 0.5f, sv2\_to\_bv2(size).y **\*** 0.5f);

b2FixtureDef FixtureDef;

*//Fixture properties*

FixtureDef.density **=** dynamic **?** 10.f **:** 0.f;

FixtureDef.friction **=** dynamic **?** 0.8f **:** 1.f;

FixtureDef.restitution **=** 1.0;

FixtureDef.shape **=** **&**Shape;

*//Add to body*

body**->**CreateFixture(**&**FixtureDef);

**return** body;

}

*// Create a Box2d body with a box fixture, from a sfml::RectangleShape*

b2Body**\*** CreatePhysicsBox(b2World**&** world, **const** **bool** dynamic, **const** RectangleShape**&** rs) {

**return** CreatePhysicsBox(world, dynamic, rs.getPosition(), rs.getSize());

}

Let’s put it to use, back to that Init() function.

*//main.cpp*

std**::**vector**<**b2Body**\*>** bodies;

std**::**vector**<**RectangleShape**\*>** sprites;

...

**void** **init**() {

...

*// Create Boxes*

**for** (**int** i **=** 1; i **<** 11; **++**i) {

*// Create SFML shapes for each box*

**auto** s **=** **new** RectangleShape();

s**->**setPosition(Vector2f(i **\*** (gameWidth **/** 12.f), gameHeight **\*** .7f));

s**->**setSize(Vector2f(50.0f, 50.0f));

s**->**setOrigin(Vector2f(25.0f, 25.0f));

s**->**setFillColor(Color**::**White);

sprites.push\_back(s);

*// Create a dynamic physics body for the box*

**auto** b **=** CreatePhysicsBox(**\***world, true, **\***s);

*// Give the box a spin*

b**->**ApplyAngularImpulse(5.0f, true);

bodies.push\_back(b);

}

}

1. **Updating physics bodies**

It’s a two steps process:

1. Stepping the physics world
2. Copying the data from the bodies to the sf::shapes

(need to render the boxes, to test they move)

1. *//main.cpp*
2. **void** **Update**() {
3. **static** sf**::**Clock clock;
4. **float** dt **=** clock.restart().asSeconds();
6. *// Step Physics world by dt (non-fixed timestep) - THIS DOES ALL THE ACTUAL SIMULATION, DON'T FORGET THIS!*
7. world**->**Step(dt, velocityIterations, positionIterations);
8. **for** (**int** i **=** 0; i **<** bodies.size(); **++**i) {
9. *// Sync Sprites to physics position*
10. sprites[i]**->**setPosition(invert\_height(bv2\_to\_sv2(bodies[i]**->**GetPosition())));
11. *// Sync Sprites to physics Rotation*
12. sprites[i]**->**setRotation((180 **/** b2\_pi) **\*** bodies[i]**->**GetAngle());
13. }
14. }
15. **Walls**

To have the boxes fall into 4 walls, add 4 walls to the init() function : their position and size will be kept into a vector that the program will loop through.

*//main.cpp*

**void** **init**() {

*// Wall Dimensions*

Vector2f walls[] **=** {

*// Top*

Vector2f(gameWidth **\*** .5f, 5.f), Vector2f(gameWidth, 10.f),

*// Bottom*

Vector2f(gameWidth **\*** .5f, gameHeight **-** 5.f), Vector2f(gameWidth, 10.f),

*// left*

Vector2f(5.f, gameHeight **\*** .5f), Vector2f(10.f, gameHeight),

*// right*

Vector2f(gameWidth **-** 5.f, gameHeight **\*** .5f), Vector2f(10.f, gameHeight)

};

*// Build Walls*

**for** (**int** i **=** 0; i **<** 7; i **+=** 2) {

*// Create SFML shapes for each wall*

...

sprites.push\_back(s);

*// Create a static physics body for the wall*

}

*// Create Boxes*

bodies.push\_back(b);

}