Physics physics physics

We are going to use a library

Lets talk about theory

We have a three js world and a physics library that will relay coordinates back to three js.

Do we want a 2d library or a 3d library? You can use a 2d library for some 3d animations as if it was seen from above. Then you only have physics for x and y axes pretty much.

There are many 3d libraries:

Ammo.js

Cannon.js

Oimo.js

Here are some 2d libraries:

Matter.js

P2.js

Planck.js

Box2D.js

There are solutions that are trying to bring physics into three js more. Like **Physijs**

**Ammo.js might be the most used library, but we will use Cannon.js because it is easy to implement and easy to get going**

**Lets import Cannon.js**

* npm install --save cannon
* import CANNON from "cannon";

This is how we install and import cannon to our project.

We are going to create the physics world so that our ball can bounce and fall.

const world = new CANNON.World();

world.gravity.set()

gravity is a Vec3 – which is similar to 3js vector3.

world.gravity.set(0, -9.82, 0);

here is how we set gravity

How do we create objects in physics world? We create bodies instead of meshes.

Lets add a sphere:

const sphereShape = new CANNON.Sphere(0.5);

where .5 is the radius

Now lets set the body

const sphereBody = new CANNON.Body({

  mass: 1,

  position: new CANNON.Vec3(0, 3, 0),

  shape: sphereShape,

});

Now we have to add the body to the world

world.addBody(sphereBody);

Nothing will happen because now we need to update our cannon.js world and our three.js world accordingly.

To update our world we have to use .step()

world.step()

in this we need to provide three params:

* A fixed time step (typically use 1/60 for 60 fps)
* How much time has passed since last step
* How much iterations the world can apply to catch up with potential delay (3 is good)

Here is what it looks like:

world.step(1 / 60, deltaTime, 3);

  sphere.position.copy(sphereBody.position);

then we copy the physics position to the real threejs position! And our sphere falls.

Now lets add the floor to our world.

floorBody.mass = 0;

by setting an objects mass to 0, it tells cannon that this object is static and will not move.

const floorShape = new CANNON.Plane();

const floorBody = new CANNON.Body();

floorBody.mass = 0;

// floorBody.position = 0;

floorBody.addShape(floorShape);

world.addBody(floorBody);

after we added the floor, the ball would just fly off towards the camera because planes are naturally created vertical and will just be a floating plane on the wrong axis. So we need to fix this.

Although rotating in CANNON is more difficult than three.js because it only support quartanion.

floorBody.quaternion.setFromAxisAngle(new CANNON.Vec3(-1, 0, 0), Math.PI \* 0.5);

here is what that looks like. We are rotating the plane on the x axis a fourth of a full rotation to make it flat. It is important to use -1 and not invert the plane. If you do, it will disappear because stuff will be below the plane technically.

Our plane in Cannon will be infinite unlike our threejs plane

We need to now add contact on material. We will create materials, unlike three js, this is for cannon to saw what the material is like that is hitting each other.

//materials for cannon

const concreteMaterial = new CANNON.Material("concrete");

const plasticMaterial = new CANNON.Material("plastic");

const concretePlasticContactMaterial = new CANNON.ContactMaterial(

  concreteMaterial,

  plasticMaterial,

  {

    friction: 0.1,

    restitution: 0.7,

  }

);

world.addContactMaterial(concretePlasticContactMaterial);

Although, nothing will happen until we add the materials to the respective objects

floorBody.material = concreteMaterial;

const sphereBody = new CANNON.Body({

  mass: 1,

  position: new CANNON.Vec3(0, 3, 0),

  shape: sphereShape,

  material: plasticMaterial,

});

Now we can simplify everything and create a default material for everything instead of concrete and plastic, etc, etc.

//converting to single material

const defaultMaterial = new CANNON.Material("default");

const defaultContactMaterial = new CANNON.ContactMaterial(

  defaultMaterial,

  defaultMaterial,

  {

    friction: 0.1,

    restitution: 0.7,

  }

);

world.addContactMaterial(defaultContactMaterial);

// Sphere

const sphereShape = new CANNON.Sphere(0.5);

const sphereBody = new CANNON.Body({

  mass: 1,

  position: new CANNON.Vec3(0, 3, 0),

  shape: sphereShape,

  material: defaultContactMaterial,

});

world.addBody(sphereBody);

//Floor

const floorShape = new CANNON.Plane();

const floorBody = new CANNON.Body();

floorBody.mass = 0;

// floorBody.position = 0;

floorBody.addShape(floorShape);

floorBody.quaternion.setFromAxisAngle(new CANNON.Vec3(-1, 0, 0), Math.PI \* 0.5);

floorBody.material = defaultContactMaterial;

world.addBody(floorBody);

here is what it looks like. We can simplify further by just setting the defaultcontactmaterial for the world to our default material and remove it from all the individual objects.

world.defaultContactMaterial = defaultContactMaterial;

Then we comment out the floorBody.material and the material: default…etc.

And the objects still interact with eachother.

There are many ways to apply forces to a [Body](http://schteppe.github.io/cannon.js/docs/classes/Body.html):

* [applyForce](http://schteppe.github.io/cannon.js/docs/classes/Body.html#method_applyForce) to apply a force to the [Body](http://schteppe.github.io/cannon.js/docs/classes/Body.html) from a specified point in space (not necessarily on the [Body](http://schteppe.github.io/cannon.js/docs/classes/Body.html)'s surface) like the wind that pushes everything a little all the time, a small but sudden push on a domino or a greater sudden force to make an angry bird jump toward the enemy castle.
* [applyImpulse](http://schteppe.github.io/cannon.js/docs/classes/Body.html#method_applyImpulse) is like [applyForce](http://schteppe.github.io/cannon.js/docs/classes/Body.html" \l "method_applyForce" \t "_blank) but instead of adding to the force that will result in velocity changes, it applies directly to the velocity.
* [applyLocalForce](http://schteppe.github.io/cannon.js/docs/classes/Body.html#method_applyLocalForce) is the same as [applyForce](http://schteppe.github.io/cannon.js/docs/classes/Body.html" \l "method_applyForce" \t "_blank) but the coordinates are local to the [Body](http://schteppe.github.io/cannon.js/docs/classes/Body.html) (meaning that **0, 0, 0** would be the center of the [Body](http://schteppe.github.io/cannon.js/docs/classes/Body.html)).
* [applyLocalImpulse](http://schteppe.github.io/cannon.js/docs/classes/Body.html#method_applyLocalImpulse) is the same as [applyImpulse](http://schteppe.github.io/cannon.js/docs/classes/Body.html" \l "method_applyImpulse" \t "_blank) but the coordinates are local to the [Body](http://schteppe.github.io/cannon.js/docs/classes/Body.html).

Because using "force" methods will result in velocity changes, let's not use "impulse" methods

Let's use **applyLocalForce(...)** to apply a small impulse on our **sphereBody** at the start:

Lets apply force to our object

sphereBody.applyLocalForce(

  new CANNON.Vec3(150, 0, 0),

  new CANNON.Vec3(0, 0, 0)

);

This says we want to apply a force on the x axis and from the middle of the object as to not rotate it at all.

Now lets add a wind on each frame in the opposite direction.

  // update physics world

  sphereBody.applyForce(new CANNON.Vec3(-0.5, 0, 0), sphereBody.position);

  world.step(1 / 60, deltaTime, 3);

we add this to the tick() function to basically add a wind force to the ball in a certain direction.

So we add the direction of the force with the -.5, and then we apply it to the sphereBody position to make it only effect the ball wherever it is.

But what about multiple objects? Here is how we will do it

We are going to create a function that will create spheres.

const createSphere = (radius, position) => {

  //three.js mesh

  const mesh = new THREE.Mesh(

    new THREE.SphereBufferGeometry(radius, 20, 20),

    new THREE.MeshStandardMaterial({

      metalness: 0.3,

      roughness: 0.4,

      envMap: environmentMapTexture,

    })

  );

  mesh.castShadow = true;

  mesh.position.copy(position);

  scene.add(mesh);

  //cannon js body

  const shape = new CANNON.Sphere(radius);

  const body = new CANNON.Body({

    mass: 1,

    position: new CANNON.Vec3(0, 3, 0),

    shape: shape,

    material: defaultMaterial,

  });

  body.position.copy(position);

  world.addBody(body);

};

createSphere(0.5, { x: 0, y: 3, z: 0 });

here is what it looks like and we have a sphere in the sky again. It is not moving because we are not updating the tick() function.

We are going to do this by creating an array. And this array is going to be of all things that need to be updated.

We start with a variable as an empty array.

const objectsToUpdate = [];

at the end of the create function we add the object we created to the array of things to update.

 objectsToUpdate.push({

    mesh: mesh,

    body: body,

  });

Then we add a loop for the array to the tick function to look through and manage all changes

 for (const object of objectsToUpdate) {

    object.mesh.position.copy(object.body.position);

  }

Now we can add more balls that can collide

createSphere(0.5, { x: 0.3, y: 3, z: 0 });

createSphere(0.5, { x: 0, y: 5, z: 0 });

or whatever we want really. Lets go back to one, and add it to lil.gui

const gui = new dat.GUI();

const debugObject = {};

debugObject.createSphere = () => {

  createSphere(0.5, { x: 0, y: 3, z: 0 });

};

gui.add(debugObject, "createSphere");

lil.gui needs an object to manipulate, so we add an object and give it the property of creating a sphere. So then we can add it to lil.gui like so and create as many spheres as we want!

Now we want to add some randomness to our sphere generator.

const gui = new dat.GUI();

const debugObject = {};

debugObject.createSphere = () => {

  createSphere(Math.random() \* 0.5, {

    x: (Math.random() - 0.5) \* 3,

    y: 3,

    z: (Math.random() - 0.5) \* 3,

  });

};

gui.add(debugObject, "createSphere");

that’s with the randomness

Now lets look at optimizing the code some.

const sphereGeometry = new THREE.SphereBufferGeometry(1, 20, 20);

const sphereMaterial = new THREE.MeshStandardMaterial({

  metalness: 0.3,

  roughness: 0.4,

  envMap: environmentMapTexture,

});

const createSphere = (radius, position) => {

  //three.js mesh

  const mesh = new THREE.Mesh(sphereGeometry, sphereMaterial);

  mesh.scale.set(radius, radius, radius);

  mesh.castShadow = true;

we move the geometry and material out side of the create function. Then we simply scale the mesh bu the passed on radius

Goiing to just need to check code for the box. It was a lot. We also added rotation to the update tick by using

object.mesh.quaternion.copy(object.body.quaternion);

in our for loop in the tick() function

Right now CANNON is testing if every object is colliding with every other object. Which is a lot of work for the cpu/gpu

This is called the broadphase. Currently CANNON is using the NaiveBroadphase which tests against every other body.

When testing the collisions between objects, a naive approach is testing every [Body](http://schteppe.github.io/cannon.js/docs/classes/Body.html) against every other [Body](http://schteppe.github.io/cannon.js/docs/classes/Body.html). While this is easy to do, it's costly in terms of performance.

That is where broadphase comes up. The broadphase is doing a rough sorting of the [Bodies](http://schteppe.github.io/cannon.js/docs/classes/Body.html) before testing them. Imagine having two piles of boxes far from each other. Why would you test the boxes from one pile against the boxes in the other pile? They are too far to be colliding.

There are 3 broadphase algorithms available in Cannon.js:

* [NaiveBroadphase](http://schteppe.github.io/cannon.js/docs/classes/NaiveBroadphase.html): Tests every [Bodies](http://schteppe.github.io/cannon.js/docs/classes/Body.html) against every other [Bodies](http://schteppe.github.io/cannon.js/docs/classes/Body.html)
* [GridBroadphase](http://schteppe.github.io/cannon.js/docs/classes/GridBroadphase.html): Quadrilles the world and only tests [Bodies](http://schteppe.github.io/cannon.js/docs/classes/Body.html) against other [Bodies](http://schteppe.github.io/cannon.js/docs/classes/Body.html) in the same grid box or the neighbors' grid boxes.
* [SAPBroadphase](http://schteppe.github.io/cannon.js/docs/classes/SAPBroadphase.html) (Sweep and prune broadphase): Tests [Bodies](http://schteppe.github.io/cannon.js/docs/classes/Body.html) on arbitrary axes during multiples steps.

The default broadphase is [NaiveBroadphase](http://schteppe.github.io/cannon.js/docs/classes/NaiveBroadphase.html" \t "_blank), and I recommend you to switch to [SAPBroadphase](http://schteppe.github.io/cannon.js/docs/classes/SAPBroadphase.html" \t "_blank). Using this broadphase can eventually generate bugs where a collision doesn't occur, but it's rare, and it involves doing things like moving [Bodies](http://schteppe.github.io/cannon.js/docs/classes/Body.html) very fast.

To switch to [SAPBroadphase](http://schteppe.github.io/cannon.js/docs/classes/SAPBroadphase.html" \t "_blank), simply instantiate it in the **world.broadphase** property and also use this same world as paramete

So the idea is to pick a better broadphase. If our objects are not moving extremely fast, then we can easily use a different broadphase to increase performance.

const world = new CANNON.World();

world.broadphase = new CANNON.SAPBroadphase(world);

doing this will increase performance, but we can also “sleep” objects that are not going to move or too far to collide. Adding the sleeper property will increase perf a lot.

const world = new CANNON.World();

world.broadphase = new CANNON.SAPBroadphase(world);

world.allowSleep = true;

world.gravity.set(0, -9.82, 0);

so we allowSleep to true and it fixes a lot of issues in frame rate.

We can also change the sleepSpeedLimit and sleepTimeLimit.

We can also listen to events and add sounds. Like collide, sleep, and wake up. Some browsers like chrome prevent sounds from playing unless the user has interacted with the page.

Lets add a sound now:

const hitSound = new Audio("/sounds/hit.mp3");

const playHitSound = () => {

  hitSound.play();

};

 body.addEventListener("collide", playHitSound);

we add this inside of our box object. We can add this to the box and sphere respectively.

We have a problem. We are asking the sound to play, but sometimes it is already playing. So, it will just continue to play after collides are already happening. Lets fix this.

const playHitSound = () => {

  hitSound.currentTime = 0;

  hitSound.play();

};

This is like reseting the sync bar on a song back to 0 unless there is something actually triggering it. Now our collide sound is too strong even if bounce is very small. So now we need the strength of the collision. If the strength is too small, we do not play the sound.

const playHitSound = (collision) => {

  const impactStrength = collision.contact.getImpactVelocityAlongNormal();

  console.log(collision.contact.getImpactVelocityAlongNormal());

  if (impactStrength > 1.5) {

    hitSound.currentTime = 0;

    hitSound.play();

  }

};

So, we get the impact velocity and test how strong it is. If it is above a certain threshold, then we make the sound. If not, then we do not make a sound.

Now we want to add some randomness to the sound.

hitSound.volume = Math.random();

we fix the randomness to scale.

const playHitSound = (collision) => {

  const impactStrength = collision.contact.getImpactVelocityAlongNormal();

  console.log(collision.contact.getImpactVelocityAlongNormal());

  if (impactStrength > 1) {

    hitSound.volume = Math.min(

      impactStrength / impactStrength,

      impactStrength / 10

    );

    hitSound.currentTime = 0;

    hitSound.play();

  }

};

Now what about removing things that go too far are leave the scene?

Lets practice this.

debugObject.reset = () => {

  for (const object of objectsToUpdate) {

    //remove body

    object.body.removeEventListener("collide", playHitSound);

    world.removeBody(object.body);

    //remove mesh

    scene.remove(object.mesh);

    //update array

    objectsToUpdate.splice(0, objectsToUpdate.length);

  }

};

gui.add(debugObject, "createSphere");

gui.add(debugObject, "createBox");

gui.add(debugObject, "reset");

This works the first time and then glitches out the second or third time you try to do it….I think there is an issue with the array… maybe use includes or something.

**Constraints**

Constraints, as the name suggests, enable constraints between two bodies. We won't cover those in this lesson, but here's the list of constraints:

* [HingeConstraint](http://schteppe.github.io/cannon.js/docs/classes/HingeConstraint.html): acts like a door hinge.
* [DistanceConstraint](http://schteppe.github.io/cannon.js/docs/classes/DistanceConstraint.html): forces the bodies to keep a distance between each other.
* [LockConstraint](http://schteppe.github.io/cannon.js/docs/classes/LockConstraint.html): merges the bodies like if they were one piece.
* [PointToPointConstraint](http://schteppe.github.io/cannon.js/docs/classes/PointToPointConstraint.html): glues the bodies to a specific point.

The component of your computer doing the physics is the CPU.

Currently, everything is single threaded and that can get overloaded quickly. The solution is to use workers.

With workers you can separate your code to different threads. We usually put the physics calcs on their own thread.

Cannon.es is being updated and better versions.