Ammo.js / Three.js Modelling Guide

Creating a motor simulation part 1
1/6/2020 Matthew Reaney

Intro

This document will explain the basics of using ammo.js / three.js to model a motor simulation without using joint constraints (these will be detailed in the next guide). It will outline some of the basic modelling concepts before utilising these to create a moving motor simulation. All the code is available here:
https://github.com/mattr862/Ammo.js-Three.js

I'd also recommend checking out the articles - https://medium.com/@bluemagnificent/intro-to-javascript-3d-physics-using-ammo-js-and-three-js-dd48df81f591 and https://medium.com/@bluemagnificent/moving-objects-in-javascript-3d-physics-using-ammo-js-and-three-js-6e39eff6d9e5.
These provide a good introduction to ammo.js / three.js and I'm using it as a basis for my code in this guide.

If you haven't used ammo.js / three.js before I've created a guide on how to get it up and running. This will leave you with a blank canvas, See the folder marked "0 Blank", all the mentioned code is found within index.html.
Starting off with the blank canvas code created from my previous guide you will see three areas marked with comments.

1. Variable declaration, this is used to define any objects.
2. Ammo.js Initialization, this initialises the libraries and simulation.
3. The start function, this equivalent to the main and runs in a loop, we use this to call functions.

Outline of motor simulation

The basic idea of creating a motor simulation is to model a static box and attach a rotatable cylinder. See figure 1. Even doing this needs to be broken down into a number of steps.

Steps to create the motor simulation

1. Create the world
   (physical properties, visuals, plane)
2. Modelling the box
3. Modelling the cylinder
4. Rotating the cylinder

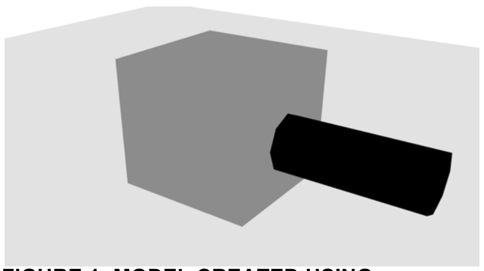
FIGURE 1: MODEL CREATED USING HTTPS://THREEJS.ORG/EDITOR/

1 Creating the World

See the folder marked “1 world”, all the mentioned code is found within index.html

Physical properties

This involves setting up the properties of your simulation and the physics that will exist within in it e.g. gravity and collision detection. See the folder mark 1 creating the world, all the mentioned code is found within its index file.

- define the world as an object variable - let physicsWorld;
- modify the properties using a function - setupPhysicsWorld()
- Call the function in our start code.

an important line of this function is –

    physicsWorld.setGravity(new Ammo.btVector3(0, -10, 0));

This defines a set force to apply to dynamic (movable) objects. For a physics world emulating Earth, the force on the y axis is set to -10. Chancing the force on the x/z can be thought of as wind.

Adding Visuals

Next we need to assign visuals e.g. background, camera, lighting.

- Add scene, camera, renderer to let physics world.
- Modify them using the functions – setupGraphics(); and renderFrame();
- Call these to functions in our start code.

Plane

Lastly, we need to add a plane to our world. This will be done by adding a rigid static body.

- Add rigidBodies = [ ], tmpTrans; to variable decaraction.
- Call tmpTrans = new Ammo.btTransform(); within start.
- create function – createBlock(); and updatePhysics();
- Call createBlock(); within start function.

Body types

There are a few different types of body that have different properties and Collison detection as well as interacting differently with the world and other objects in it.
Shape descriptors -

- Rigid – This defines the mesh of an object that can’t be changed.
- Softbody – This allows the shape of the object to change.

Physics descriptors

- Static –completely fixed/motionless but still objects can still collide with it.
- Dynamic – responds to physics within the world.
- Kinematic – responds only to changes made by user input.

createBlock function

The Create block is a good general template for creation of objects with the world. It handles the visuals and physics of the block object in three stages.

1. In the first section of its code it defines the main parameters of the block. Position, scale and mass. Note the mass in our example is 0, this means that the objected will be unaffected by forces/collisions.
2. The three.js section creates the shape of the block using a mesh
3. The ammo.js section defines the physical properties and collision type.

The world should now be as shown in figure 2.

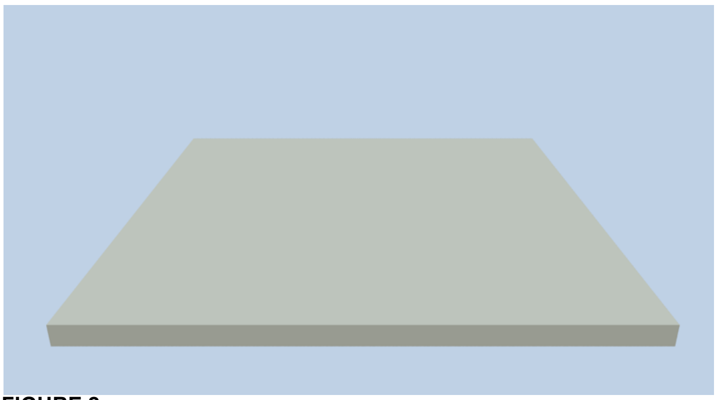
FIGURE 2

2 Modelling the Box

See the folder marked “2 box”, all the mentioned code is found within its index file.

As the box is the motor body, we don’t want it to move. This means implementing it as a rigid static body similar to the plane in the last section.

- create function – createBox();
- Call createBox(); within start function.

We can simply alter the position and scale arrays within the first section of the createBox function. To create a new box. For this shape I’m using –

```
let pos = {x: 0, y: 3, z: 0};
let scale = {x: 5; y: 5, z: 5};
```

In the three.js section I changed the colour to a darker shade of grey by simply altering the hexadecimal colour value. –

```
let blockPlane = new THREE.Mesh(new THREE.BoxBufferGeometry(), new
THREE.MeshPhongMaterial({color: 0x9ca3ad}));
```

I would recommend trying the online three.js editor - https://threejs.org/editor/
It has a much nicer GUI with changes to properites it works the same why by altering the pos and scale arrays.

The world should now be as shown in figure 3.

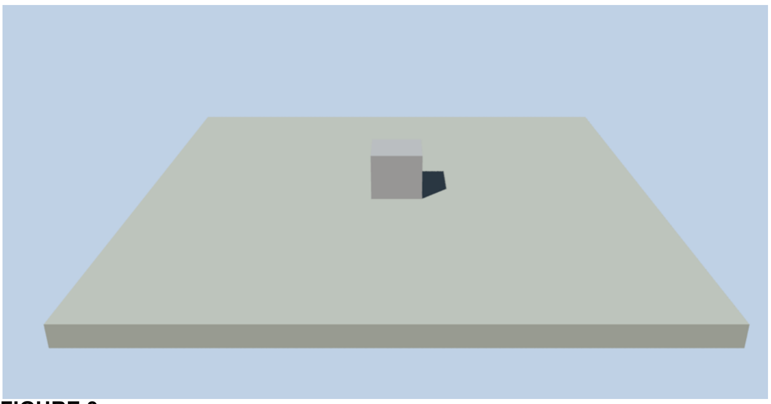
FIGURE 3

3 Modelling the Cylinder

See the folder marked "3 Cylinder", all the mentioned code is found within index.html

Unlike the box and plane the cylinder requires the ability to move with user input however we don't want it to respond to external physics so we can implement it as a rigid kinematic object. However, the first few steps will be similar.

- Create function – createCyliner()
- Call createCyliner(); within start function.

Create Cylinder function

This is similar to before, but a few properties must be changed and added. For this shape I'm using –

```
let pos = {x: 5, y: 3, z: 0};
let scale = {x: 1; y: 5, z: 1};
```

Within the three.js section we change "BoxBufferGeometry()" to "CyclinerBufferGeometry(1, 1, 1, 8)" and I'm changing its colour to black (hex value 0x000000)This gives us the shape in figure 4. Next we need to alter the shapes rotation to it will attach as shown in figure 1.

To do this we add –

```
Cylinder.rotation.x = 1.571;
Cylinder.rotation.z = 1.571;
```

Note three.js works in radians (1.571 radians = 90 degrees).

This gives us the shape in figure 5.

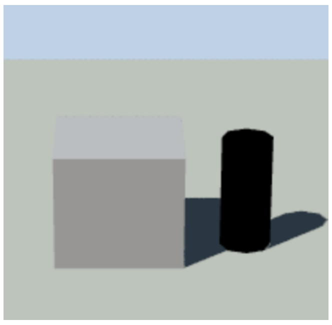
FIGURE 5

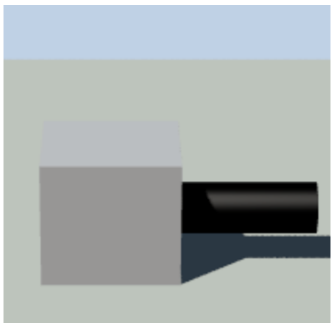
FIGURE 4

Similar to the three.js model we need to update the physics mesh from box to cylinder Changing “btBoxShape” to “btCylinderShape”.

Implementing Kinematic object type

Setting the object up as a kinematic is the hard part. To do this we need to add event flags to change the motion state of the object between motionless and moving with user input motion.
Firstly, add all of these to the variable declaration section -

```
let Cylinder = null,
const FLAGS = { CF_KINEMATIC_OBJECT: 2 }
const STATE = { DISABLE_DEACTIVATION : 4 }
```

These control the motion states of the object. As the cylinder object is going to be manipulated from other functions it requires global declaration.

Next we need to add a few more physics definitions that the very bottom of the createCylinder function.

```
body.setFriction(4);
body.setRollingFriction(10);
body.setActivationState( STATE.DISABLE_DEACTIVATION );
body.setCollisionFlags( FLAGS.CF_KINEMATIC_OBJECT );
Cylinder.userData.physicsBody = body;
```

The world should now be as shown in figure 6.

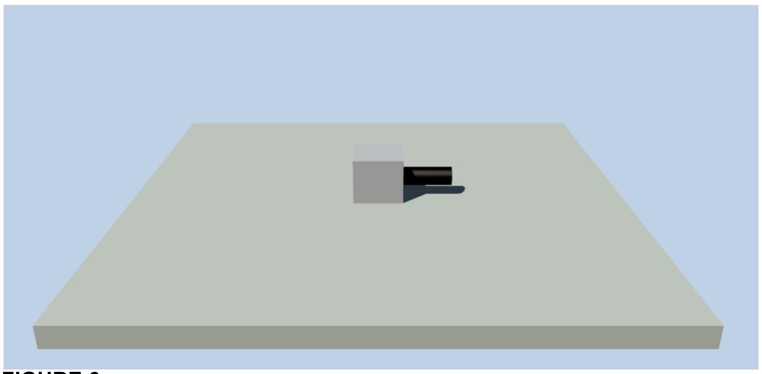
FIGURE 6

4 Rotating the cylinder

See the folder marked “4 rotation”, all the mentioned code is found within index.html

To implement the rotation of the cylinder we are going to control it using the up and down arrow keys. To allow for user input with these buttons we must set up 3 functions. Placed just after render frame function.

1. setupEventHandlers – this calls the next two functions
2. handleKeyDown(event) – this reads a button being pressed down
3. handleKeyUp(event) – this reads a button being released / not pressed

The handleKey functions utilise switching statements to check each key press. Add more actions performed by key presses is as simple as adding more cases. Just ensure any objects being manipulated are declared as global.

In HandleKeyDown - Case 38 represents pressing the up arrow down. And rotates the cylinder by using the code -   Cylinder.rotation.x += 0.175 // +10 degrees
case 40 represents pressing the down arrow down. And rotates the cylinder the opposite direction by using the code -   Cylinder.rotation.x -= 0.175 // -10 degrees

In HandleKeyUp - Both cases are now blank this means no rotation is preformed when the button is released.

Lastly, we then need to call setupEventHandlers before render frame in start function.

Conclusion

Upon running the final version of the code, you should be able to see a flicker of movement when holding down the keys. That's the motor working! In the next guide I'm going to build on the simulation by exploring joint constraints. I've listed below a number of helpful resources for the topics discussed in the document.

Resources:

Intro to JavaScript 3D Physics using Ammo.js and Three.js - https://medium.com/@bluemagnificent/intro-to-javascript-3d-physics-using-ammo-js-and-three-js-dd48df81f591 and https://medium.com/@bluemagnificent/moving-objects-in-javascript-3d-physics-using-ammo-js-and-three-js-6e39eff6d9e5.

Guide to three.js modelling and visuals - https://discoverthreejs.com/book/first-steps/shapes-transformations/

Three.js website - https://threejs.org specifically the page with object rotation https://threejs.org/docs/#api/en/core/Object3D.rotation

Github thread on object rotation - https://github.com/mrdoob/three.js/issues/910

Web article on keyboard input object rotation - https://subscription.packtpub.com/book/web_development/9781783981182/1/ch01lvl1sec22/adding-keyboard-controls

Three.js tutorial site, specifically transformations section - https://imagecomputing.net/damien.rohmer/teaching/2018_2019/semester_1/m2_mpri_cg_viz/tutorial/content/012_transformations/index.html