# Motor Modelling in Ammo.js / Three.js

Part of a QLab project for creating a robot arm simulation
16/6/2020 Matthew Reaney

## Introduction

This document will explain the basics of using ammo.js / three.js to model a motor simulation using joint constraints. 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. For this specific document you will want to use code from the motor modelling folder.

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.

## Recommendations

I'd also recommend checking out the demos found here: https://github.com/schteppe/ammo.js-demos and 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 a lot of code in this guide. At the end of the document there is also a large references section detailing all the articles and forum posts that I found useful for this project.

# Contents



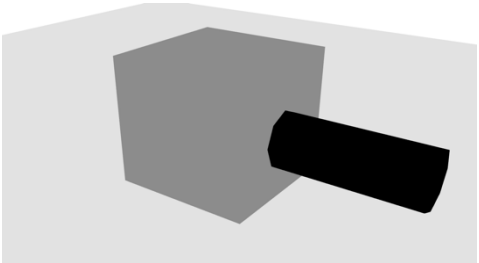

**FIGURE 1: MODEL CREATED USING HTTPS://THREEJS.ORG/EDITOR/**

# 1. Creating the World

The completed code for this section is stored within the folder marked “1 world”

## 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.

## Outline of 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.

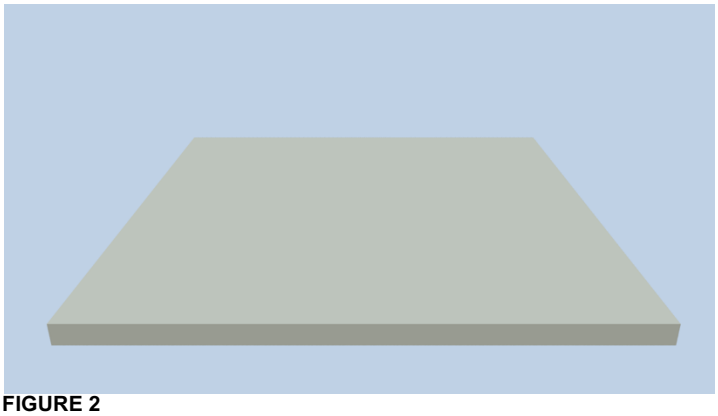

**FIGURE 2**

# 2. Modelling the Motor Body

The completed code for this section is stored within the folder marked “2 box”

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 for seeing changes to properties. It works in the same as the code by altering the corresponding pos and scale arrays.

The world should now be as shown in figure 3.

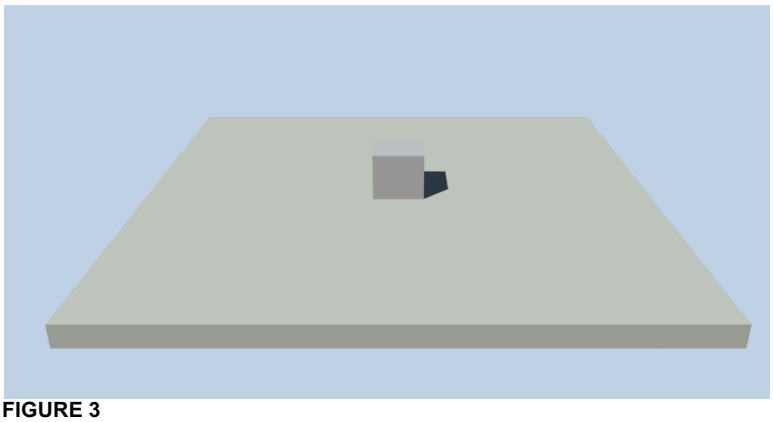

FIGURE 3

# 3. Modelling the Motor Cylinder

The completed code for this section is stored within the folder marked “3 Cylinder”

Unlike the box and plane the cylinder requires the ability to move and respond to external physics so we will implement it as a rigid dynamic 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 alter its quat array to

```
let quat = {x: 0, y: 0, z: 1, w: 1};
```

and add the line

```
Cylinder.rotation.set(quat.x, quat.y, quat.z, quat.w);
```

This gives us the shape in figure 5.

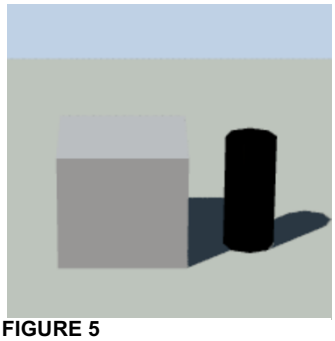

FIGURE 5

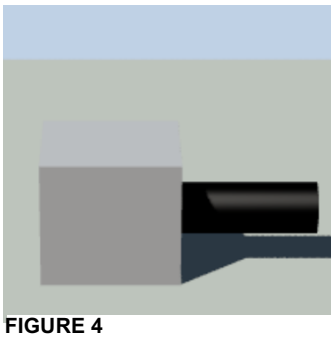

FIGURE 4

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

## Implementing Dynamic object type

Setting the object up as a Dynamic requires some extra work. As we want the object to be affected outside of its modelling function we have to define it as a global object in the variable objects section as well as declaring a dynamic motion state flag. Firstly, add all of these to the variable declaration section -

```
let Cylinder = null
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.

```
cylinderBody.setFriction(4);
cylinderBody.setRollingFriction(10);
cylinderBody.setActivationState( STATE.DISABLE_DEACTIVATION );
physicsWorld.addRigidBody( cylinderBody );
Cylinder.userData.physicsBody = cylinderBody;
rigidBodies.push(Cylinder);
```

The world should now be as shown in figure 6. Note as the cyclinder is down under the effects of gravity it falls to the ground.

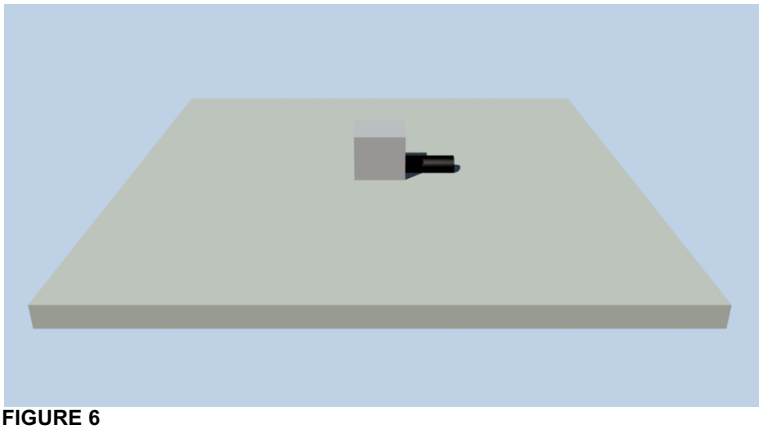

**FIGURE 6**

# 4. Joint constrainting the Motor.

## Outline of joint constraints

There are four main types of joint constraints:

1. Point to point (p2p) – this is the simplest method and works by connecting two bodies at a single point allowing for joint translation and can be used to daisy chain many objects together. A good application is a ball and socket joint.
2. Hinge – This locks to objects together by limiting their relative rotation to the axis of the hinge joint. This is useful for doors or wheels as it limits rotation to one axis.
3. Slider – allows the bodies to translate in only one axis this can be useful in parts that extend like a piston.
4. Cone Twist – This is a special application of point to point and allows rotation on the cone axis while still linking translation.

More Info on joints and diagrams are available in this article:
https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/Maya-SimulationEffects/files/GUID-CDB3638D-23AF-49EF-8EF6-53081EE4D39D-htm.html

## More on P2P constraints

For our model I'm going to be using a p2p constraint to attach the cylinder to the box. The general formula for a p2p requires:

- Two sets of body information for the two objects being joined. These are set up in the Ammo.js section.
- Two sets of pivot points. These should be set up after the Ammo.js using the properties of the object using the objects data.
- A declaration of the joint using the four previous pieces of information.
- Adding the joint to the physics world

There are different ways of varying complexity of meeting all the requirements for a p2p:

1. Local Variables (simple) – This method involves using only one function, by setting up both objects, the pivots and constraints locally. This works well if you won't need to later alter the objects or constrains.
2. Global Variables (complex) – This method requires three functions, one for each object and their respective pivots and one to declare and add the joint. It requires global declaration at the top of the code for the four sets of data.
3. You can mix the two by using global variables but calling it within an already existing function, that's what I intend to do for this model.

## Implementation

The completed code for this section is stored within the folder marked "4 Joint"

We now want to joint constrain the box to the cylinder, as mentioned above we are going to implement this by using method 3.

Firstly, we need to globally declare our four variables:

```
let boxBody = null, cylinderBody = null, boxPivot = null, cylinderPivot = null
```

Next we need to alter the createBox() function:

- Change "let body = …" to "boxBody = …"
- And change "physicsWorld.addRigidBody( body );" to "physicsWorld.addRigidBody( boxBody );"
- At the bottom of the function add

```
boxPivot = new Ammo.btVector3(5, 0, 0 );
```

Next we need to do the same for the  createCylinder function:

- Change “let body = …” to “cylinderBody = …”
- Anywhere else you see the term “body” in this function replace it with “cylinderBody”
- At the bottom of the function add

```
cylinderPivot = new Ammo.btVector3( 0, 0 , 0);
```

Lastly, we need to define and enable the joint. This can be done in a separate function called after the creation of the shapes or at the end of whitherever create function is called last. So at the end of createCylinder add:

```
let p2p = new Ammo.btPoint2PointConstraint( cylinderBody, boxBody, cylinderPivot, boxPivot);
physicsWorld.addConstraint( p2p, true );
```

The first line creates/declares the joint and the second activates it in the world.

The world should now be as shown in figure 7.

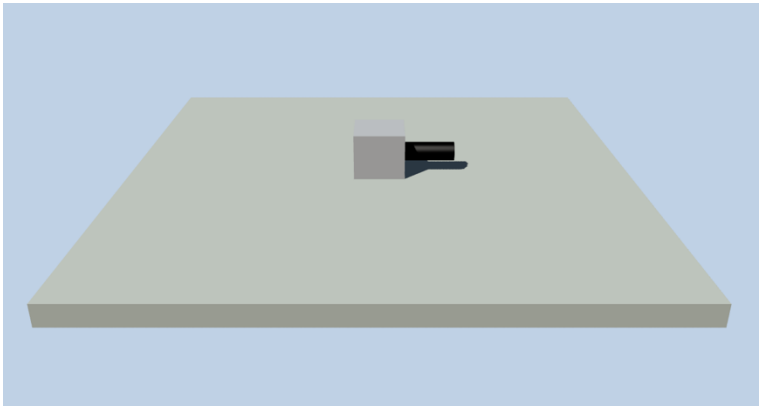

**FIGURE 7**

# 5. Rotation via Keyboard Input

The completed code for this section is stored within the folder marked “5 Rotation”

To implement the rotation of the cylinder we are going to control it using a function which calculates the forces each frame based on input from the keyboard up and down arrow keys.

## User input via the Keyboard

To allow for user input with these buttons we must set up three 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

Firstly, we need to declare a global array so data can be shared between multiple functions by button presses.

```
let moveDirection = { up: 0, down: 0 }
```

Next we need call setupEventHandlers function in the start function before renderFrame, as far as I can tell these work similar to hardware interrupts, instead of constantly checking for events it essentially updates the system (global variables) when an event happens. In our case the events are button presses or release.

Lastly, we need to set up the event cases using the handleKey functions. These utilise switching statements to check each key press or release. Adding more actions performed by key presses is as simple as adding more cases, just ensure any objects being manipulated are declared as global.

## moveCylinder function

Now we have the keyboard input setup we need to create a function that uses the moveDirection array to apply a force to rotate the cylinder.

```
function moveCylinder() {
      let scalingFactor = 1; // changes speed of rotation
      let resultantImpulse = new Ammo.btVector3( moveDirection.up -
      moveDirection.down, 0, 0 ) // rotation vector
      resultantImpulse.op_mul(scalingFactor);
      let physicsBody = Cylinder.userData.physicsBody;
      physicsBody.setAngularVelocity( resultantImpulse ); }
```

We can do this by utilising the **.setAngularVelocity** function to rotate the cylinder with an input resultant force bu using uses the **resultantImpulse** and a **scaling factor**. The resultantImpulse is calculated by using the global variable controlled by keyboard input and simply indicates which axis and direction the cylinder spins on and the scaling factor essentially controls the strength/speed of rotation.
This is different to the more standard .setLinearVelocity function which moves the on the x,y,z axis, using the same resultant impulse and scaling factor combination.

Lastly call the moveCylinder function within the renderframe function before updatePhysics**, not in the start function**. This means the forces and motion will get updated before the frame is rendered and directly the simulation steps to the frame rate.

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!

# Conclusion

A finalised version of the code for this project with additional comments and fixes can be found under the folder marked “motor sim”. This concludes this document, however, work on the project will be continued in future documents. As the documentation of Ammo.js / Three.js is sparse, I’ve listed below a number of additional resources that I found useful for the topics discussed in the document.

# Resources

Ammo.js / three demos with code: https://github.com/schteppe/ammo.js-demos
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
Info on joints and diagrams available in this article - https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/Maya-SimulationEffects/files/GUID-CDB3638D-23AF-49EF-8EF6-53081EE4D39D-htm.html
A forum thread I visited a lot while working on these topics - https://forum.playcanvas.com/t/solved-trying-to-build-a-robot-arm-grabber-using-ammo-js/9979
and a selection of code from a joints demo - https://github.com/kripken/ammo.js/blob/master/ammo.idl#L618
Thread discussing object rotation - https://stackoverflow.com/questions/27363299/how-to-rotate-an-object-in-bullet-physics
Programming bullet constraints - https://docs.panda3d.org/1.10/python/programming/physics/bullet/constraints
Rotation with quat array - https://studiofreya.com/game-maker/bullet-physics/proper-rotation-with-quaternions-with-bullet-physics/