**Getting started with three.js and ammo.js**

**Introduction**

Welcome to my guide on ammo.js and three.js this guides aims to give you the knowledge on the fundamentals of using these libraries and act as a Lego kit of sorts for building your own personal projects.

**What is three.js and ammo.js**

Ammo and three.js are both JavaScript libraries, three.js is a graphics engine which allows you to create and display animated 2d graphics in a web browser using webGL, this is a great library with many user and great projects supporting it so it is not hard to learn these resources include the three.js website <https://threejs.org/> which contains several examples of what can be done with he library and has documentation explaining all the code and what it does. Ammo.js is a physics engine which allows you to create a physics world which can be used in conjunction to a graphics engine to create graphical physics simulations. This library is a port of the popular c and c++ bullet physics engine therefore it is a niche library with very few guide and project using it, that is why this guide exists, I am to show you how to use this engine.

**There are 4 main part of ammo.js concepts that you should be aware of to get use of this engine these are:**

- Rigid bodies

- Soft bodies

- Collision detection and filtering

- Constraints

**Rigid bodies**

Bodies in ammo.js are called collision objects or more commonly rigid bodies. Rigid bodies are the objects in the simulation which move, collide, and have mass and can have impulses applied to it.

There are three types of rigid bodies and many body shapes can be implemented in ammo.js. The three types of ammo.js bodies are:

- Static Rigid bodies – These bodies have a fixed position throughout the physics simulation and cannot be moved when defining these bodies they have a mass of 0 applied to them.

- Kinematic Rigid bodies – These bodies are not affected by the physics of the Ammo.js world but are can be animated during the physics simulation, These bodies are similar to the static rigid bodies in that they are have a mass of 0, but unlike the static rigid bodies we add the bodies to the rigid bodies array we typically define globally at the start of our program script to allow the positions of it and its three.js counterpart to be changed in the worlds as it is animated.

- Dynamic rigid bodies – These are the most intensive rigid body type in ammo.js, these bodies are fully affected by the physics of the world.

**Soft bodies**

Soft bodies are defined as bodies in computer graphics that visually and realistically represent the motion and properties of deformable objects while being able to some extent retain its shape, these bodies can represent real life objects balloons or cloths.

**Constraints**

Constraints can be thought of as joints that would be used real life to connect multiple objects together. There are multiple types of constraints in ammo.js to carry out different functions:

Point – to – point constraints.

Point to point constraints limit the translation of two pivot points of rigid bodies to match the world space. Using this constraint, you can create a chain of rigid bodies. This constraint can be useful for creating something like a robotic arm or in the case of my structures project a plank of wood using several rigid bodies in column and rows.

Hinge constraints

Hinge constraints are joints which restrict rotation around the pivot of two bodies to only one axis, therefore this axis could be useful for creating something like a door. Or even flaps on an airplane model as this constraint also allows the user to specify the limits and motor of the hinge.

Slider constraint

The slider constraint allows a body to rotate around one axis and translate along that axis. This constraint would be useful for creating something like a piston.

Cone twist constraints

This cone constraint is a special version of the point-to-point constraint that adds cone and twist axis limits. The axis for this constraint serves as a twist axis. This constraint is useful for creating something like a ragdoll.

**Collision detection**

Collision detection in ammo.js is a concept that allows for the collision and interaction between objects, There are four main concepts that aid with collision detection which allow you to show how objects interact in a world.

- Contact Manifold check

- Contact test

- Contact pair test

- Ghost objects

To understand each of these I suggest you check out this tutorial: <https://medium.com/@bluemagnificent/collision-detection-in-javascript-3d-physics-using-ammo-js-and-three-js-31a5569291ef>. As collision detection is a big subject in ammo.js we will only be covering the basics for this tutorial.

**Getting started**

Before we start programming these concepts you need to first set up your work space to do this you will need to install ammo.js and three.js from this link: when you are done that you can create a folder with whatever project name you want and inside that folder create an index file and a js folder containing both the ammo.js/ three.js files, for a more in depth tutorial on setting up your workspace you can use this link: <https://github.com/mattr862/Ammo.js-Three.js/blob/master/Setting%20up%20ammo.js%20Three.js.pdf>.

Now that you have set up your folder you will need to also have an ide which supports javascript and html if you do not have this already I suggest you install visual studio code which is what will be used during this how-to guide: <https://code.visualstudio.com/download>.

Now you are ready to start programming, with your index.html file open in the ide you should now do the “! Tab” short cut at the top of the document to auto create your html page. Now you will want to import the libraries you will be using for this body by using the script tag in body and src from the root of the folder to “js/ammo.js” and “js/three.js” and create another script tag which will contain your JavaScript code for this tutorial.

Your code should look like this:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Setting up world</title>

</head>

<body>

    <script src="js/three.js"></script>

    <script src="js/ammo.js"></script>

    <script>

    </script>

</body>

</html>

### **In your script tag you will also want to create a Init function ( init short for initiate ) which will contain all the main function calls and an enter the statement Ammo().then( Init ); which will initiate ammo.**

After this you need to declare the global variables for the program, so at the top of the script we we enter the following code:

//variable declaration

        let physicsWorld, controls, scene, camera, renderer; //three.js variables

let rigidBodies = []; // ammo.js variables

### **Declaring variables and setting up graphics**

**Before creating the physics world, we need to first create the graphics by creating a world three.js which will be what you see on the browser, To do this we define a setUpGraphics function which instantiates the three main attributes for our three.js world: "the camera, renderer, and scene". We will define all these attributes at the beginning of the script which the world will be based on so their values can be accessed globally. There are many other attributes you can add like lighting and so on but we will not be covering these in this tutorial. In this function, we will instantiate also a clock attribute which will be used later when we begin working with ammo.js.**

  function setupGraphics()

        {

            //create clock for timing

            clock = new THREE.Clock();

            //create the scene

            scene = new THREE.Scene();

            scene.background = new THREE.Color( 0xbfd1e5 );

            //create camera

            camera = new THREE.PerspectiveCamera( 17, window.innerWidth / window.innerHeight, 0.2, 5000 );

            camera.position.set( 50, 180, -170 );

            camera.lookAt(new THREE.Vector3(0, 0, 0));

            //Add hemisphere light

            let hemiLight = new THREE.HemisphereLight( 0xffffff, 0xffffff, 0.1 );

            hemiLight.color.setHSL( 0.6, 0.6, 0.6 );

            hemiLight.groundColor.setHSL( 0.1, 1, 0.4 );

            hemiLight.position.set( 0, 50, 0 );

            scene.add( hemiLight );

            //Add directional light

            let dirLight = new THREE.DirectionalLight( 0xffffff , 1);

            dirLight.color.setHSL( 0.1, 1, 0.95 );

            dirLight.position.set( -1, 1.75, 1 );

            dirLight.position.multiplyScalar( 100 );

            scene.add( dirLight );

            dirLight.castShadow = true;

            dirLight.shadow.mapSize.width = 2048;

            dirLight.shadow.mapSize.height = 2048;

            let d = 50;

            dirLight.shadow.camera.left = -d;

            dirLight.shadow.camera.right = d;

            dirLight.shadow.camera.top = d;

            dirLight.shadow.camera.bottom = -d;

            dirLight.shadow.camera.far = 13500;

            //Setup the renderer

            renderer = new THREE.WebGLRenderer( { antialias: true } );

            renderer.setClearColor( 0xbfd1e5 );

            renderer.setPixelRatio( window.devicePixelRatio );

            renderer.setSize( window.innerWidth, window.innerHeight );

            document.body.appendChild( renderer.domElement );

            renderer.shadowMap.enabled = true;

        }

### **Creating the Physics world**

**Now we create the physics world using Ammo.js. This is a separate world from the three.js world which has physics properties and will be the basis for what the objects we create in the three.js world is going to be doing in each frame.**

  function setupPhysicsWorld()

        {

            let collisionConfiguration  = new Ammo.btDefaultCollisionConfiguration(),

                dispatcher = new Ammo.btCollisionDispatcher(collisionConfiguration),

                overlappingPairCache = new Ammo.btDbvtBroadphase(),

                solver = new Ammo.btSequentialImpulseConstraintSolver();

            physicsWorld = new Ammo.btDiscreteDynamicsWorld(dispatcher, overlappingPairCache, solver, collisionConfiguration);

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

        }

### **Creating the render function**

**The next step is creating a render frame function, this function will create a render loop which for every frame the scene and camera are updated. However, you may notice that we only use the renderer in this loop meaning that we only render the three.js world without considering the ammo.js world. This is because at the minute there is no need to add physics to the three.js world. After all, there are currently no objects to apply physics to. We will cover this in the creating objects part of this tutorial.**

 function renderFrame()

        {

            renderer.render( scene, camera );

            requestAnimationFrame( renderFrame );

        }

### **Function calls**

**Now in the init function created earlier you should invoke all the functions we have created so far so that the program works.**

        function Init()

        {

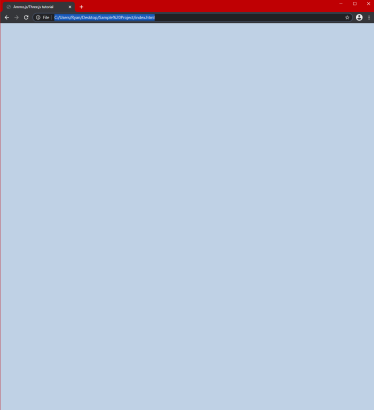
            //code goes here

            setupPhysicsWorld();

            setupGraphics();

            renderFrame();

        }

**Now if you open the live server in vs code you should see just a blue screen this is good it means your code is working right now. The reason that there is nothing in it is because we have not added anything but the physics and the graphics world itself. If you don’t get a blue screen at this point like the image below there is an issue and you should go back and check your code.**

**Rigid bodies**

**Static rigid body**

**The first body we are going to create is a static rigid body we do this by creating both a cube in ammo.js and three.js with a long x and z-axis and short y-axis( x represents the width of the object, z represents length and y represents height, these three values x,y,z are known as vector3 in programming and can be used to set the position of an object, size, and rotation) and a mass of 0, this mass of 0 will make the body static. This will create the world in which your program will be based.**

 function createBlock()

        {

            let pos = {x: 0, y: 0, z: 0};

            let scale = {x: 50, y: 2, z: 50};

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

            let mass = 0;

            //threeJS Section

            let blockPlane = new THREE.Mesh(new THREE.BoxBufferGeometry(), new THREE.MeshPhongMaterial({color: 0x3c4145}));

            blockPlane.position.set(pos.x, pos.y, pos.z);

            blockPlane.scale.set(scale.x, scale.y, scale.z);

            blockPlane.castShadow = true;

            blockPlane.receiveShadow = true;

            scene.add(blockPlane);

            //Ammojs Section

            let transform = new Ammo.btTransform();

            transform.setIdentity();

            transform.setOrigin( new Ammo.btVector3( pos.x, pos.y, pos.z ) );

            transform.setRotation( new Ammo.btQuaternion( quat.x, quat.y, quat.z, quat.w ) );

            let motionState = new Ammo.btDefaultMotionState( transform );

            let colShape = new Ammo.btBoxShape( new Ammo.btVector3( scale.x \* 0.5, scale.y \* 0.5, scale.z \* 0.5 ) );

            colShape.setMargin( 0.05 );

            let localInertia = new Ammo.btVector3( 0, 0, 0 );

            colShape.calculateLocalInertia( mass, localInertia );

            let rbInfo = new Ammo.btRigidBodyConstructionInfo( mass, motionState, colShape, localInertia );

            let body = new Ammo.btRigidBody( rbInfo );

            physicsWorld.addRigidBody( body, colGroupPlane, colGroupPlankBlocks );

        }

**Dynamic rigid body**

**Now we are going to create a dynamic rigid body, this type of body is created similarly to how a static rigid body is creating the main difference is that it has mass to physics can be applied to it, and as well as this we add is to our rigid Bodies array that we defined earlier this is so we can use this array later so we can update the positions correlating three.js objects of the ammo.js bodies so that we can see the effects of the physics world in the browser for this example we are creating a sphere object instead of a cube object that we used in for making** the plank object in the first example.

  function createBall(){  // This is just a example function for creating an object in ammo

            let pos = {x: 0, y: 20, z: 0};

            let radius = 2;

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

            let mass = 1;

            //threeJS Section

            let ball = new THREE.Mesh(new THREE.SphereBufferGeometry(radius), new THREE.MeshPhongMaterial({color: 0xff0505}));

            ball.position.set(pos.x, pos.y, pos.z);

            ball.castShadow = true;

            ball.receiveShadow = true;

            scene.add(ball);

            //Ammojs Section

            let transform = new Ammo.btTransform();

            transform.setIdentity();

            transform.setOrigin( new Ammo.btVector3( pos.x, pos.y, pos.z ) );

            transform.setRotation( new Ammo.btQuaternion( quat.x, quat.y, quat.z, quat.w ) );

            let motionState = new Ammo.btDefaultMotionState( transform );

            let colShape = new Ammo.btSphereShape( radius );

            colShape.setMargin( 0.05 );

            let localInertia = new Ammo.btVector3( 0, 0, 0 );

            colShape.calculateLocalInertia( mass, localInertia );

            let rbInfo = new Ammo.btRigidBodyConstructionInfo( mass, motionState, colShape, localInertia );

            let body = new Ammo.btRigidBody( rbInfo );

            physicsWorld.addRigidBody( body, colGroupPlankBlocks, colGroupPlane );

            ball.userData.physicsBody = body;

            rigidBodies.push(ball);

        }