**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++ bulett 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 librarys 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.