Advanced Ammo.js / Three.js Guide

1/6/2020 Matthew Reaney

Introduction

This is a series of individually useful functions for use in ammo.js and three.js with basic explanations of how they look and work. All the code can be found in the advanced folder in my GitHub: https://github.com/mattr862/Ammo.js-Three.js
If you haven't worked with GitHub before I'd recommend checking out the other guides on my GitHub too.

Contents

Iterative shapes

The completed code for this example is stored within the folder marked "Iterative"

The ability to create shapes using an iterative function isn't necessarily for complex modelling purposes but for creating an environment within the world for objects to engage with. For example, a large array of unconstrained blocks creates the destruction of one large destructible wall.

To create iterative code, it's not too difficult the main premise is to put any modelling code into a "for" loop (other iterative loops should work too). If we then nest the 'for" loop, we can not only create a line of shapes but then a wall of shapes and nesting again crates a 3d area of these shapes.

We also need to add or alter a few things in our modelling code.

1. We need to add iterative values for controlling the x,y,z directions. I used blocksX, blocksY and blocksZ.
2. Altering the pos vector for the blocks by using the iterative counter variable(s).
   ```
   let pos = {x: i - (blocksX/2 ), y: j+1, z: k};
   ```
   Note the x value is I – blocksX/2 this positions the blocks in a central location.
3. As the blocks are stacked end to end, we need to remove the code that adds rolling friction to the blocks. This variable causes lots of collisions between the blocks, this causes the shapes to sporadically collapse and the code to run slowly

As a general rule the more shapes the more unstable the construct becomes.

As an example I've put simple 1x1x1 cubes into a 2 different formations.
figure one : `let blocksX = 4, blocksY = 4, blocksZ = 4;`
figure two : `let blocksX = 10, blocksY = 5, blocksZ = 1;`

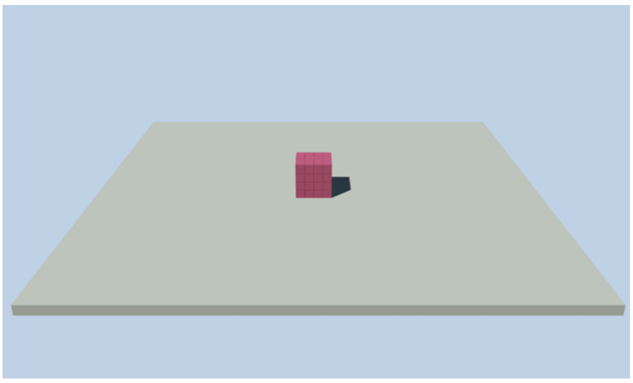


FIGURE 1

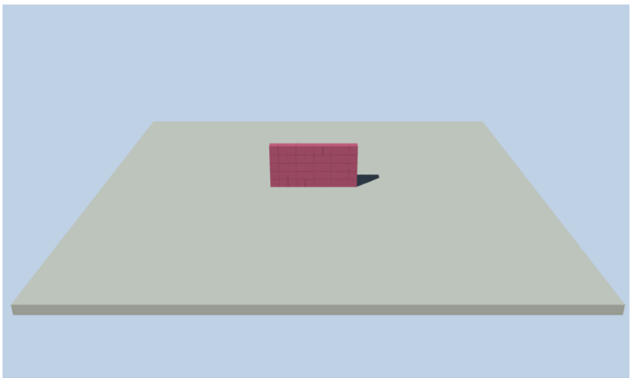


FIGURE 2

Invisible shapes

The completed code for this example is stored within the folder marked "Invisible"

The main reason to use invisible shapes is to restrict the movement of other objects within the world via an inviable barrier. This can be important for a number of reasons, such as inability to have object's move off surfaces or preventing objects from falling forever. This can cause issues with collisions as well as slowing down the simulation because calculations are constantly being made.

To implement an invisible shape, we simply implement a rigid static object in ammo.js but unlike previous modelling we don't apply any visuals using three.js. To show possible uses for invisible shapes I'm going to create two different example scenarios utilising the same iterative shapes code as above.

I've set up a scenario in which a number of the blocks will fall off the left of the screen. Take a look at the "Baseline" to see this without any invisible barriers. For my first example (figure 3) I have implemented an invisible floor, much like the baseline the blocks fall off the left of the screen however they simply land and stop slightly further down as seen in figure 3. For my second example (figure 4) I have implemented an invisible wall on the left side of the screen preventing any blocks from falling off the plane as seen in figure 4.

Apart from the removal of the three.js code the shape is also defined by having a one for the plane on which it restricts movement and the other two planes have a zero. Much like mass a zero here actually means this plane goes on infinitely.

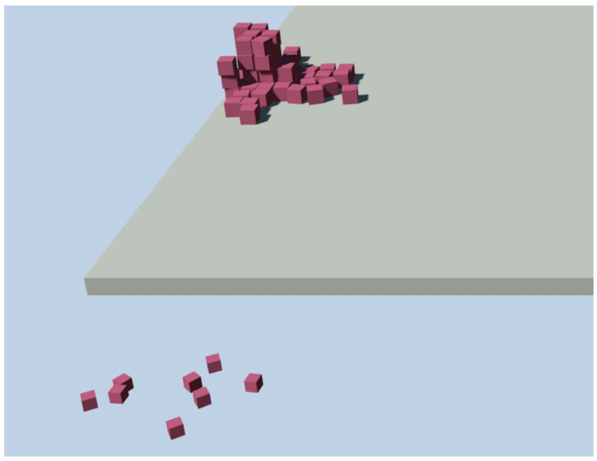


FIGURE 4

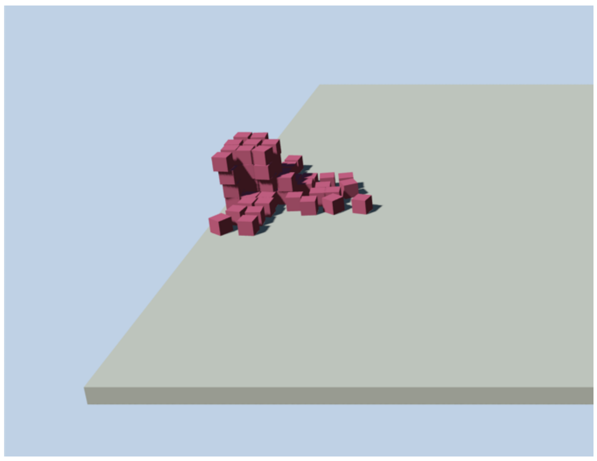


FIGURE 3

Camera control

The completed code for this example is stored within the folder marked “Camera”

It’s often a problem that you don’t get a good view of physics playing out from the stationary camera set so wrote some code to allow for modification of the cameras angle and position using the keyboard. This code doesn’t interact with the world or physics in anyway but is useful for debugging physics situations, inspecting models, physics interactions and getting screenshots.

The code functions similarly to the rotation of the cylinder in one of the examples on the previous worksheets. It uses a set of event handlers to trigger interrupts. I used two sets of global variables. One for the direction (angle) and one for the position. These are then manipulated within the events then feed into the camera function. The key lines of code run in each event are:

```
    camera.position.set( camPos.x, camPos.y, camPos.z );

    camera.lookAt(new THREE.Vector3( camDirection.x,
camDirection.y, camDirection.z));
```

Also note only the button pressed events need to be used. The button release section is left completely blank.

Below I’ve listed all the event case tied to button press for this code.

Case 13 : Enter key : Resets camera to original position
Case 37 : Left arrow : Moves camera left
Case 38 : Up arrow : Moves camera up
Case 39 : Right arrow : Moves camera right
Case 40 : Down arrow : Moves camera down
Case 65 : A : Rotates camera left
Case 68: D : Rotates camera right
Case 83 : S : Rotates camera down
Case 87 : W : Rotates camera up
Case 187 : + / = : moves camera away
Case 189 : - / _ : moves camera closer

Below I’ve taken some screen shots to demonstrate the ability to move the camera:

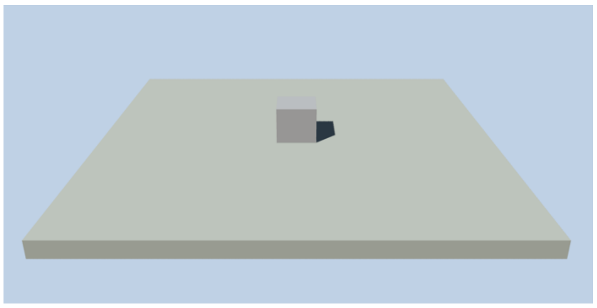


FIGURE 5: ORIGONAL

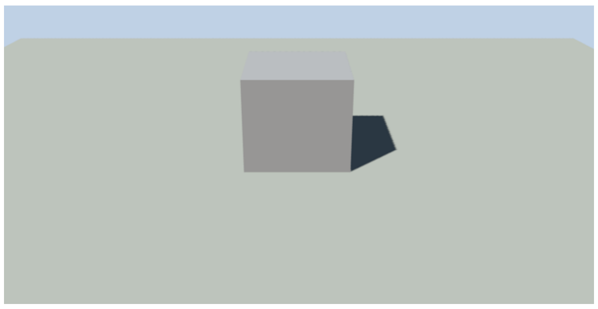


FIGURE 6: MOVE CLOSER

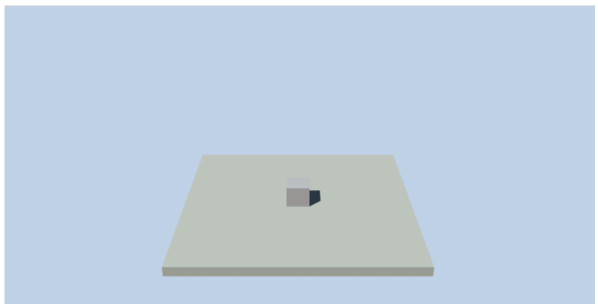


FIGURE 7: MOVE AWAY

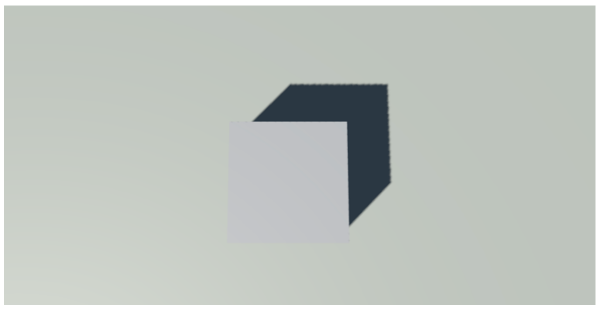


FIGURE 8: VIEW FROM ABOVE

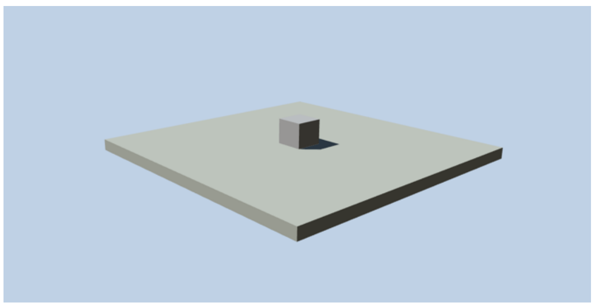
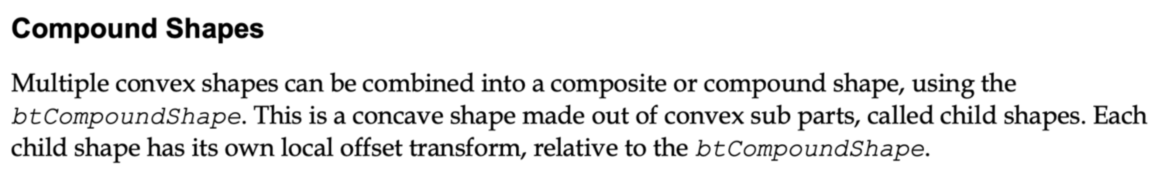


FIGURE 9: VIEW FROM THE SIDE

Compound shape creation

Bullet manual mentions compound shapes under the heading of collision detection, page 19 http://www.cs.kent.edu/~ruttan/GameEngines/lectures/Bullet_User_Manual



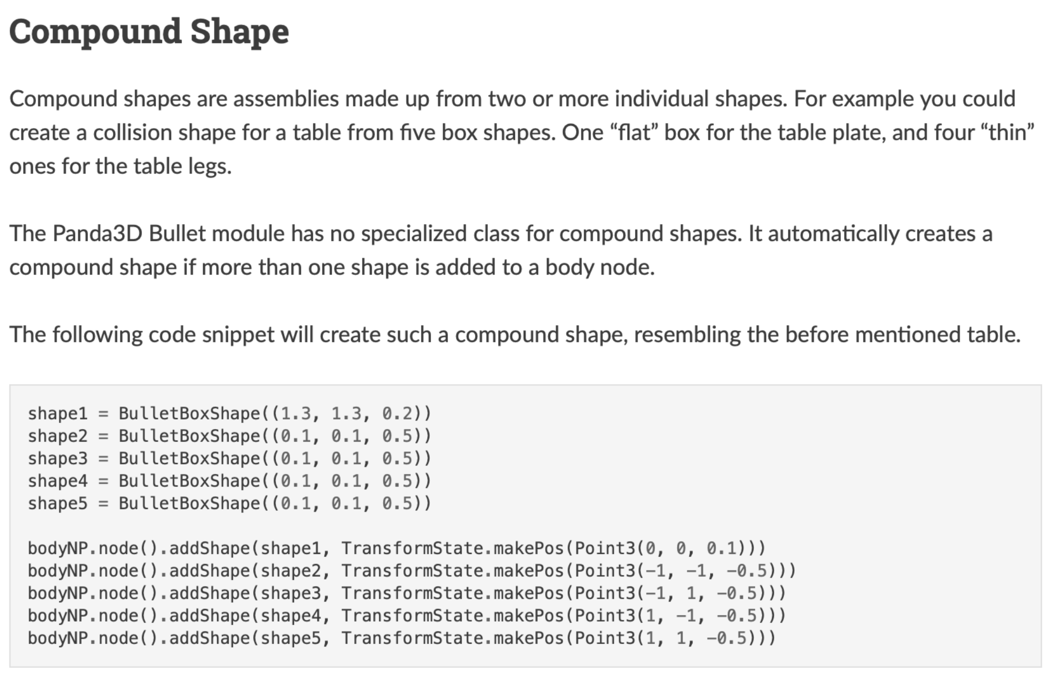
Compound Shapes

Multiple convex shapes can be combined into a composite or compound shape, using the *btCompoundShape*. This is a concave shape made out of convex sub parts, called child shapes. Each child shape has its own local offset transform, relative to the *btCompoundShape*.

Looking at the demo code for vehicle as it mentions complex shapes. It makes use of some sort of special object type in ammo called a `btRaycastVehicle`. I think this allows storage of vehicle data alongside a model and in this case is using a compound shape.
Starts off with chassis then adds four child shapes (wheels) on individually.

This webpage sort of explains the concept in its introduction:
https://subscription.packtpub.com/book/game_development/9781783281879/7/ch07lvl1sec44/compound-shapes

Article mentions building a compound shape table in bullet code, converting this to ammo could be a good way to experiment with compound shapes.
https://docs.panda3d.org/1.10/python/programming/physics/bullet/collision-shapes



Compound Shape

Compound shapes are assemblies made up from two or more individual shapes. For example you could create a collision shape for a table from five box shapes. One “flat” box for the table plate, and four “thin” ones for the table legs.

The Panda3D Bullet module has no specialized class for compound shapes. It automatically creates a compound shape if more than one shape is added to a body node.

The following code snippet will create such a compound shape, resembling the before mentioned table.

```
shape1 = BulletBoxShape((1.3, 1.3, 0.2))
shape2 = BulletBoxShape((0.1, 0.1, 0.5))
shape3 = BulletBoxShape((0.1, 0.1, 0.5))
shape4 = BulletBoxShape((0.1, 0.1, 0.5))
shape5 = BulletBoxShape((0.1, 0.1, 0.5))

bodyNP.node().addShape(shape1, TransformState.makePos(Point3(0, 0, 0.1)))
bodyNP.node().addShape(shape2, TransformState.makePos(Point3(-1, -1, -0.5)))
bodyNP.node().addShape(shape3, TransformState.makePos(Point3(-1, 1, -0.5)))
bodyNP.node().addShape(shape4, TransformState.makePos(Point3(1, -1, -0.5)))
bodyNP.node().addShape(shape5, TransformState.makePos(Point3(1, 1, -0.5)))
```