The first problem I had to deal with was creating my own transformation component to override unity. First I created 3 Vector 3s: Scale, Rotation and Transformation. For scale vector you place the x to z diagonally in the matrix with the 4th row being zero which creates the scale Matrix, to create the translation matrix you need to create an identity matrix with the 4th row being the translation vector. Rotation is more complicated as it the rotations come in as degrees from the vector 3 you must convert them to radians otherwise the rotation matrix will be inaccurate. To do this you simply multiply the vector 3 by PI / 180. You then must convert the Euler to a quaternion, first you create trigonometry identities of the roll pitch and yaw using cos and sin. You then combine Cos Pitch and Cos Yaw by multiplying it and do the same with Sin Pitch and Sin Yaw. Then these Identities are multiplied or added together to give you a new quaternion. You then need to convert this quaternion to a matrix.

Once you have the 3 Matrices you then multiply them together in an order known as TRS or Translation Rotation Scale. If the order is wrong and you rotate first the movement of the object will be incorrect and so will the scale. The order is highly important for the matrices. To multiply the matrices together you must multiply the rows by the Colum for each value of new matrix. So for a 2 by 2 Matrix top left value would be Matrix 1[0, 0] \* Matrix 2 [0, 0] + Matrix 1[1, 0] \* matrix 2 [0, 1]. The Matrix is then multiply by a vector 3 which is the model space vertices, however when unity converts this vector 3 to a vector 4 the w value is 0 causing an issue in the multiplication and needs to be set to 1 in the code, the multiplication is simple with the pattern being you multiply ROW x column 0 by the Model space .x, row x column 1 for y and so forth.

The Major problem was the physics, physics in games is hard to represent accurately without computational issues my physics system was kept simple to not bulk to CPU. By using methods like additional and multiplication the player can apply a force to an object, this force is always added by Gravity \* mass. To multiply a vector 3 you multiply the x by x, y by y and z by z. As the simulation does not have atmosphere I do not have to run certain mathematical equations solving issues with air resistance and fluid dynamics being ran on the CPU. The force is then applied to acceleration by dividing the force by mass, to divide a vector 3 you divide the x by x, y by y and z by z. However this does create an issue, with having no air-resistance simulation an object with 200 mass will fall the same as an object with 1 mass and while this accurate Neil Armstrong proved on the Moon I would or preferred higher mass objects falling faster than lighter objects.

The Acceleration is then multiplied by time (Delta time) and added to velocity. The velocity is then added to the objects position multiplied by time again which creates a semi implicit physics system. This system is done on a fixed update which is called less frequently then updates preventing tunnelling occurring with collision detection.

The final problem I was faced with was collision detection and resolution, for collision objects I have AABB and Bounding Sphere. With AABB Collision detection I used the Christer Ericson formula finding it was more accurate and reduced lag in my game the formula also allowed me to do resolution which is what I am going to focus on as it is the harder concept to implement. To implement collision resolution you need to have a normal and penetration, you then work out the centre and half’s of the two boxes. To work out the penetration and normal you must add the box 1 x to the box 2 x and then subtract that by the absolute of box 1 centre x – box 2 centre x . This is now the current penetration which is checked against penetration which at first is max value of float, if true the penetration is set to the current penetration. I then check if the box is to the right or left by checking if the box 1 centre x is more than box 2 centre x and setting the normal to either be 1, 0, 0 or -1, 0, 0 depending if it’s on the right or left, you repeat these steps for y and z which allows you to accurately push an object away without too much force. You need to add on a value to push otherwise the object will always be colliding this value for me was small enough to stop jittering but still allow for physics.