Final Year Module
Physics Simulation

University of Teesside

Sam Oates
J9060283

Summary of implemented scenarios

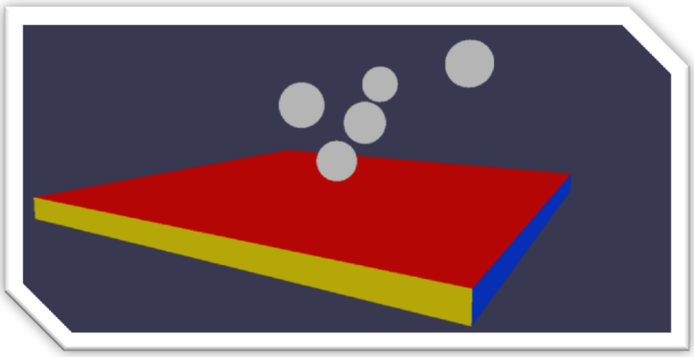
I aimed to implement multiple scenarios of physics based simulations in games. The aim was to take a given scenario and brake said scenario down into a simplified version to extrapolate the data needed to create advanced simulations. I created six test scenarios covering the following physics simulations;

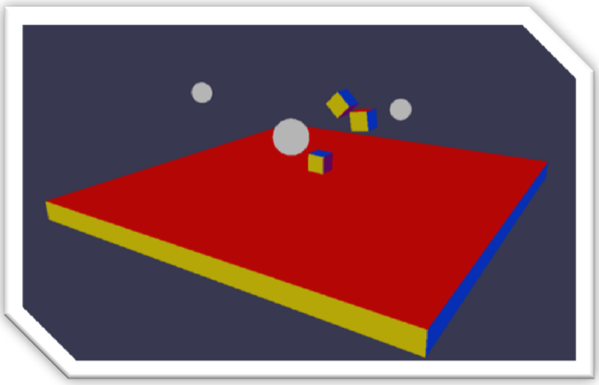
- Sphere on sphere collision (such as a pool and or snooker based game)
- General collisions and impulses (multiple dynamic boxes and dynamic spheres, as well as a static ground box)
- See-saw simulation
- Sphere rolling down a slope
- Knocking down stacked boxes with a sphere
- Simple box on box collision

Sphere on sphere collision

The sphere on sphere collision scene tests five dynamic spheres each of different mass falling onto one and other above a static rigid body box.

The simulation uses sphere on sphere collision as well as sphere on axis aligned bounding box (AABB).





General collisions and impulses

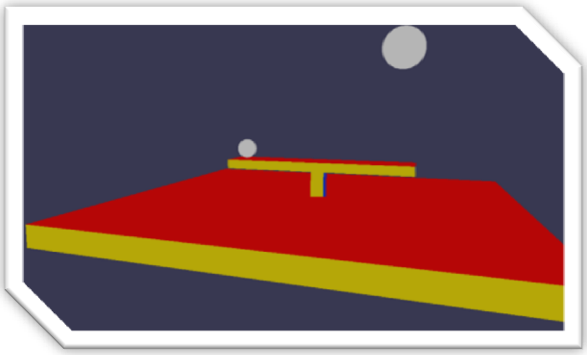
The reason for this simulation was to test each different collision type against one and other, as well as testing different masses and sizes of the same collision type.

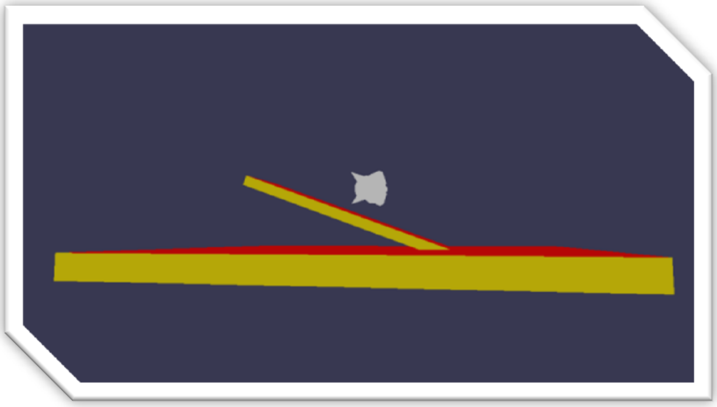
This simulation tests sphere on sphere, sphere on AABB, AABB on sphere and AABB on AABB, as well as simulating static and dynamic rigid body types.

See-saw simulation

The see-saw simulation requires the use of object-orientated bounding boxes (OBB), allowing the dynamic plank to pivot on the static box hinge as the spheres fall upon it.

The simulation was created to best simulate applying impulse to a dynamic rigid body from the angular velocity of another dynamic rigid body.





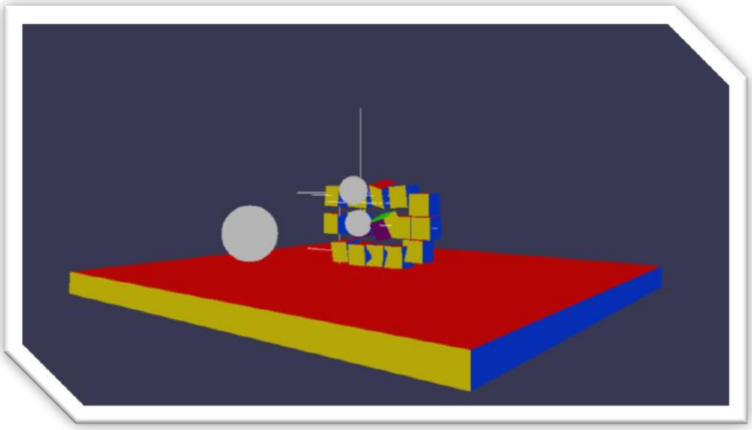
Sphere rolling down a slope

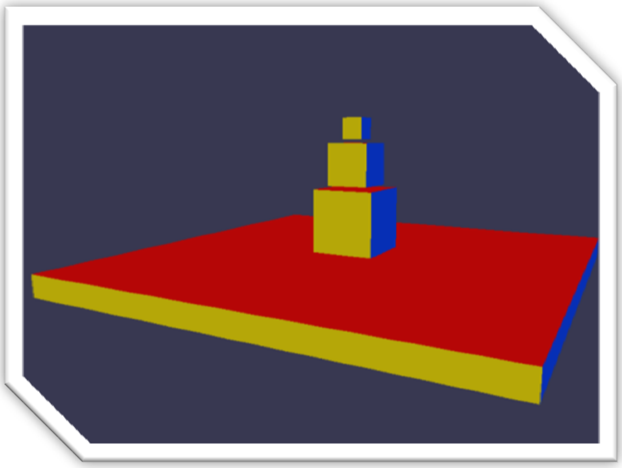
To test torque, angular momentum and angular velocity this test places a mesh with a sphere collider at the top of an angled static rigid box body.

This simulation uses sphere on OBB collision detection, taking the point of collision and collision tangent to apply toque to the rigid body sphere.

Knocking down stacked boxes with a sphere

The simulation launches two dynamic spheres into a 5 by 3 set of stacked rigid body boxes. The dynamic spheres are dropped onto a static sphere, also testing sphere on sphere impulse and collision normal calculation. The simulation also has the most number of dynamic objects, resulting in this simulation being used for stress testing.





Simple box on box collision

This basic simulation environment is used for testing the following collision types; AABB on AABB, OBB on AABB, AABB on OBB and OBB on OBB.

This simulation is used as a basis for testing dynamic rigid body boxes under the influence of the force gravity.

The rigid body implementation overview

Rigid bodies within my simulation react to collision impulses as well as having the ability to apply a force and or torque to a given point upon the rigid body's surface. Torque is accumulated per rigid body update and reset at the end of each update. The net torque is then applied to the angular momentum of the rigid body. Following the updating of the inverse inertia tensor of the rigid body, angular momentum is combined with the updated inverse inertia tensor to give angular velocity. Using angular velocity I form a skew matrix from which the orientation quaternion is formed.

$$\varpi = \begin{matrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{matrix}$$

Figure 1: Skew Matrix

Applying torque at a point

Applying torque at a given point on a rigid body, takes a world space coordinate of the point to apply a force and the vector of the force to apply. Taking the world coordinate, we transform it into the rigid body's local space by subtracting the world space away from the rigid body's centre of mass. We then take the given force and cross product it with the newly calculated local space coordinate, adding the result to the body's stored torque.

Calculating angular momentum and velocity

Angular momentum is calculated from the net torque applied to a given rigid body multiplied by delta time. Angular momentum represents the product of a body's rotational inertia and rotational velocity about a given axis, where net torque represents the rate of change in angular momentum.

$$\mathrm{I}_t = \mathcal{R}_t \mathrm{I}_{body} \mathcal{R}_{t^T}$$

Figure 2: Inertia Tensor Calculation

To calculate the angular velocity of a rigid body, we take the angular momentum and multiply it by the body's inverse inertia tensor. The inertia tensor can be calculated from the body's inertia tensor when the body is at rest and the rotation matrix of the body. The inverse of the inertia tensor is calculated in the same manor, but using the inverse inertia tensor at the body's rest. Finally we can calculate the angular velocity vector, by multiplying the inverse inertia tensor with the angular momentum.

Calculating the rotational skew matrix

$$\begin{bmatrix} 0 & -\varpi_z(t) & \varpi_y(t) \\ \varpi_z(t) & 0 & -\varpi_x(t) \\ -\varpi_y(t) & \varpi_x(t) & 0 \end{bmatrix}$$

Figure 3: The angular velocity skew matrix, where ω represents angular velocity.

To rotate the render-able object represented by an Ogre scene node we need to transform the angular velocity into a quaternion. The first step to achieve this is to create a rotational skew matrix from the angular velocity vector. Once we have the angular velocity skew matrix we can update the body's rotational matrix. To update the rigid body's rotational matrix, we take the current rotational matrix and increment it by the product of the current rotation matrix, the skew matrix and delta time. Once we have the updated rotational matrix, we can update the quaternion used to represent the orientation of the rigid body.

Strengths and weaknesses of my simulation

The linear part of impulses within my simulation holds up well; both accuracy wise as well as performance wise. Along with this, I believe how my simulation loads scene data and allows easy setup of scenes and real-time modifications of scenes to be a crucial part of debugging issues I have encountered along the way.

The simulation itself suffered from errors within collision detection, rather than starting with simple sphere on sphere detection I attempted to implement OBB on OBB detection. Errors in the calculation of the collision data (normal, position and penetration) resulted in hours of debugging the physics code itself (which turned out to have little to no error). This was later rectified and the collision problems were solved by adding the rendering of collision bounds and collision normals to help with debugging.

My simulation has errors within rotational impulse calculation, although the impulse and update code seems to be correct. This again, is most likely caused by errors within collision data calculation, resulting in over rotation sometimes occurring within the simulations.

Overall I am extremely pleased with the setup of my application, the code structure and component like behaviour of the code, allowing easy expansion in the future. In theory the well-structured code should also make it easier to pin point errors within the physics calculations themselves, should they occur.