

# Custom physics demonstration & explanation

The physics simulation demonstrates collision between dynamic bodies when balls collide with each other and when the spring launcher hits a ball, dynamic-kinematic collision when a ball collides with a surface or flipper, forces being applied to dynamic bodies when a ball passes through a booster or hits a bouncer (yellow sphere), and spring joints in the ball launcher mechanism.

## Collision Detection

Collision detection is done in *PhysicsScene* every timestep after each object is updated. *checkForCollision* iterates over each object, then checks for a collision between it and every actor after it, using the shape ID of the *PhysicsObjects* to determine what collision detection method to use by indexing an array of functions with *(ID1 \* shape count) + ID2*, where shape count is the number of different shape IDs, giving every permutation of shape IDs a unique index. A consequence of this is that sphere-box and box-sphere, for example, are considered unique collisions, so separate functions exists for both situations, with one of them calling the other function and reversing the order of the parameters, so while there are two functions for a given situation, only one of them contains the solution.

There are a total of 6 unique collisions that can occur: plane-plane, plane-sphere, plane-box, sphere-sphere, sphere-box, and box-box. The one exception is plane-plane, which will always return false as planes are always kinematic, and therefore do not need to collide. This leaves 5 distinct collision detection functions.

Plane-sphere collisions are detected by projecting the sphere onto the planes normal and checking if the distance between the plane and the sphere is less than the spheres radius. If it is, they are overlapping. In addition to this, the direction the sphere is moving in is checked relative to the planes normal is checked, because planes only block collisions in one direction. If both checks pass, the sphere is colliding with the plane.

Plane-box collisions are detected by iterating over each of the boxes corners and projecting them onto the planes normal to find the distance between them and the plane. The velocity of each point (combined velocity and angular velocity) is also projected onto the planes normal to determine if each point is moving towards the plane. If both checks pass for any point, the box is colliding with the plane.

Sphere-sphere collisions are detected by finding the distance between the two spheres and subtracting it from the sum of their radii. The resulting value is the penetration depth, and if its greater than 0, the spheres are colliding.

Sphere-box collisions are detected by transforming the spheres position into the boxes local space by first subtracting the boxes position, then projecting it onto the boxes local x and y direction vectors. This allows the box to be treated as an AABB. Using the spheres relative position, the closest point on the box to the sphere can be found by taking the relative position and constraining it within the box’s extents through checking if the components are greater than the extent or less than the negative extent, and if so, setting it to that value. This closest point is then transformed back into world space by reversing the operation done to get the local position. The distance between the sphere and the point on the box closest to the sphere is then found by subtracting the point from the spheres position, then finding the length of the resulting vector. The penetration distance is the spheres radius minus the distance to the closest point, and if the value is greater than 0, the sphere is colliding with the box.

Box-box collisions are detected using the separating axis theorem. The *Box* class contains the *checkBoxCorners* function, which determines if another boxes corners are inside of it, and provides the contact point, penetration, and collision normal via reference parameters if so. The function iterates over each corner of the other box, transforming them to be relative to the boxes local space. The corner point is used to find the projection of the other box in local space to check for a separating axis, and it is checked if it is within the extents which would mean the corner is inside the box. After iterating over each corner, the projection of the other box is checked against the extents to find a separating axis. If one exists or none of the corners are inside the box, the function returns false. The penetration distance is found by getting the minimum penetration from the projection axis. The axis this is found on is also used to determine the collision normal. To determine if a collision has occurred, this function is called on both boxes, and if the penetration is greater than 0, the boxes are colliding.

## Collision Resolution

When a collision detection function determines the objects are colliding, it will call *resolveCollision* on one of the objects to resolve the collision. There are two implementations for *resolveCollision*, one for plane-rigidbody collision, and another for rigidbody-rigidbody collision. The plane implementation is similar the rigidbody implementation, except it assumes the plane has infinite mass and no velocity, so the entire impact is applied to the rigidbody, and no force is applied to the plane.

The rigidbody-rigidbody implementation of *resolveCollision* is implemented in *Rigidbody*, taking a pointer to the other rigidbody, the contact point, the collision normal, and the penetration distance. The first thing done is to check if either object is a trigger, in which case it will add the other object into its list *objectsInsideThisFrame* and call *triggerEnter* on it. When one of the objects being collided with is a trigger, no collision resolution is done, allowing other objects to pass through it. The distance between the collision point and the centre of each object is found as its radius. The centre of the object is found using the virtual function *getCenter* in *Rigidbody*, which returns the objects position by default, but can be changed for objects that rotate off-centre, such as flippers. The relative speeds of the contact points are found by getting the dot product of the velocity and collision normal to get the speed toward the other object, and the angular velocity of the contact point by multiplying the radius by the object’s angular velocity. If the objects are moving away from each other, the collision will resolve on its own and the function exits, otherwise it will continue to push them away from each other. At this point, it calculates the effective mass of the collision point of each object using *point mass = 1 / (1 / mass + radius2 / moment of inertia)*. This considers both the mass of the object and the moment of inertia when applying a force at the contact point. The force applied to each object is found by *force = (1 + e) \* m1 \* m2 / (m1 + m2) \* (speed1 – speed2) \* collision normal*, where *e* is the average elasticity of the objects, *m1* and *m2* are the point masses of each object, and *speed1* and *speed2* are the relive speeds of the contact points. The force is applied to each object using the function *applyForce*, taking a force and the position its applied at, being the contact point. After this, the collision callback is called on each object, and the static function *applyContactForces* is called, passing a pointer to both objects, the collision normal, and the penetration distance.

A contact force is used to prevent objects from being inside each other by moving them apart by the amount they are inside each other. The function *applyContactForces* takes a pointer to two rigidbodies, the collision normal, and the penetration distance. It first checks if either object is a trigger, and exits if so. It then checks if the second actor is a null pointer or is kinematic, where it uses the maximum value of an int for its mass and will otherwise use its actual mass. In the case of an object colliding with a plane, only one object is passed, so the second object will be a null pointer. Using this, the factor each object is moved by is determined by *obj2mass / (obj1mass + obj2mass)*. The first object is then moved using *setPosition*, to its current position minus the product of the previously mentioned factor, the collision normal, and the penetration depth, while the second object uses *1 – factor* and adds the product to its position.

A rigidbody can be used as a trigger using the function *setTrigger*, which causes other objects to stop colliding with it, and instead have the public function pointers *onTriggerEnter*, *onTriggerExit*, and *whileInsideTrigger* called and passed the related physics object.

Springs are a *PhysicsObject* that tries to keep two rigidbodies at a set distance from each other by applying forces to them. When creating a spring, the constructor takes two rigidbody pointers for the objects to connect, a spring coefficient, dampening, a resting length between the objects, and connection points on the objects. In *fixedUpdate*, the spring finds the force to apply to the objects by *spring coefficient \* (rest length – current length) – dampening \* relative velocity of objects*. The effect of this is that the further the two objects are from the rest length and the larger the spring coefficient, the larger the force bringing the objects back to the rest length will be, and the larger the dampening, the more it will resist them moving apart.

# Improvements

Currently, collision is checked between every object in the scene, regardless of how far away it is. Spatial partitioning could be used to reduce the number of collision checks by storing objects within quadrants, with each quadrant storing objects near each other. The result is that collision only needs to be checked against objects within the same quadrant. A simpler method of achieving a similar result is checking the distance between two objects is less than the combined approximate radius of the two objects. This could be implemented by adding a virtual *getCollisionRadius* function to *PhysicsObject* that is overridden by each class to return some approximate radius. For example, *Sphere* could return its radius plus a constant, and *Box* could return the sum of its extents.

When an actor is removed from a *PhysicsScene* by an actors *fixedUpdate*, a null reference exception will be encountered due to *PhysicsScene.update* using a range based for loop. Changing this to handle actors being removed or changing how *addActor* and *removeActor* to add and remove objects at the end of an update cycle would provide more freedom in managing *PhysicsObjects*.

If an object is moving fast enough, its possible that it could be on one side of an object on the current time step, and on the other side of an object in the next time step, allowing an object to tunnel through other objects. Using half-steps or quarter-steps during collision detection would help prevent this by moving an object a fraction of the distance it will travel during the time step and checking if it will collide in that position, and if not, continue moving it until it reaches its final position.

# Third party libraries

Bootstrap was used to display the graphics of the application.

# References

<http://jonathanbosson.github.io/reports/TNM085_group5.pdf>

<https://en.wikipedia.org/wiki/List_of_moments_of_inertia>

<https://gamedev.stackexchange.com/questions/69369/angular-velocity-from-2d-collision>

https://stackoverflow.com/questions/2259476/rotating-a-point-about-another-point-2d