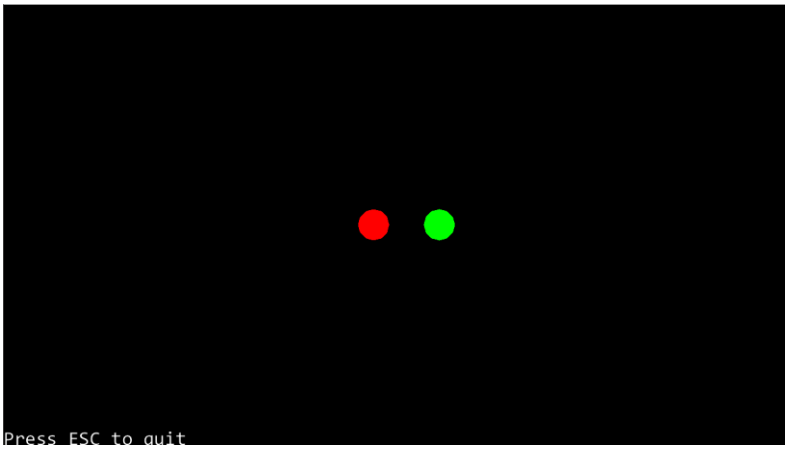
Tutorial – Linear Force and Momentum

Introduction and Objective:

In this tutorial we will add a *Sphere* class to our simulation, which extends the *PhysicsObject* abstract class we implemented in the previous tutorial.

We will also implement some basic movement, collision detection, and collision response, so that we can give our spheres some initial velocity and watch them collide.



Prerequisites:

You will need to have completed the following tutorials:

- Fixed Timestep Tutorial – available under the *Introduction to Physics* session.

Physics Objects:

For our purposes, you can assume that each physics object (or actor) in the scene has a single collision volume and that there are initially only three types:

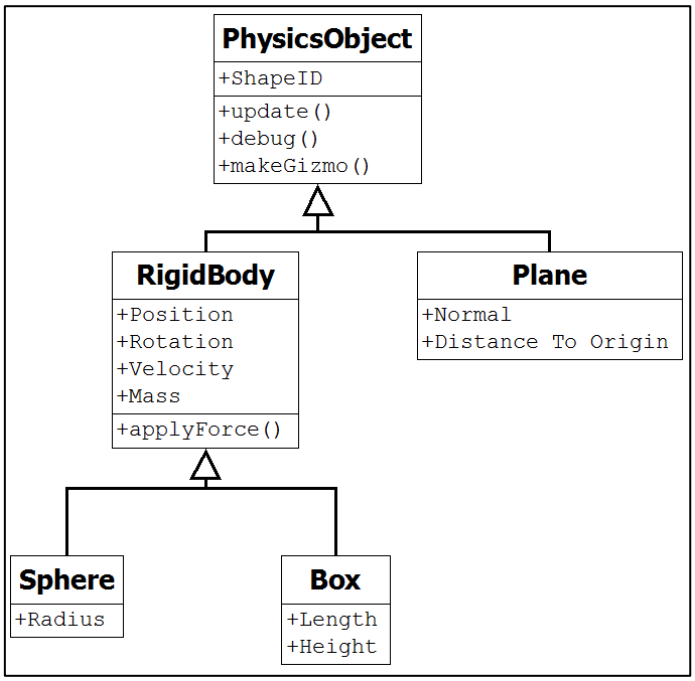
- Plane
- Sphere
- Box

The actors in a physics engine can be static or dynamic. Static objects do not move so only need position and rotation (remember this is 2D so we need a vec2 for position and single float for the rotation angle). The Plane can only be static (it makes no sense for something with infinite extents and infinitely small thickness to move about in our scene)

Dynamic actors have some additional properties:

- Mass
- Velocity

The following UML diagram illustrates the complete actor class hierarchy that we will implement in our physics engine:



Note how everything is derived from a single base class, the *PhysicsObject* class we implemented in the previous tutorial. This design choice was made so that we can iterate through a single list when updating and adding gizmos to our scene. This will also allow us to write our collision detection routines in a neater form later on.

In this tutorial we've updated the *PhysicsObject* class to include a single variable: ShapeID. This is used for collision detection in a later tutorial. The code will be neater if you use an enumerated type for this.

Your updated definition for the *PhysicsObject* class is as follows:

```
enum ShapeType {
    PLANE = 0,
    SPHERE,
    BOX
};

class PhysicsObject {
protected:
    PhysicsObject(ShapeType a_shapeID) : m_shapeID(a_shapeID) {}

public:
    virtual void fixedUpdate(glm::vec2 gravity, float timeStep) = 0;
    virtual void debug() = 0;
    virtual void makeGizmo() = 0;
    virtual void resetPosition() {};

protected:
    ShapeType m_shapeID;
};
```

You won't need a 'getter' to return the *m_shapeID*.

In this tutorial we will only create the *RigidBody* and *Sphere* sub-classes. You may wish to start defining the *Box* and *Plane* class yourself now, although we will cover these in a future tutorial.

The RigidBody Class:

All dynamic actors will inherit from the *RigidBody* class. Because this class will serve as a base class for any kind of dynamic actor (sphere, box, or any other shape that may be added in the future), it will encapsulate the variables relating to movement and position.

Create the new *RigidBody* class and update the class definition as follows:

```
class Rigidbody : public PhysicsObject {
public:
    Rigidbody(ShapeType shapeID, glm::vec2 position,
            glm::vec2 velocity, float rotation, float mass);
    ~Rigidbody();

    virtual void fixedUpdate(glm::vec2 gravity, float timeStep);
    virtual void debug();
    void applyForce(glm::vec2 force);
    void applyForceToActor(Rigidbody* actor2, glm::vec2 force);

       virtual bool checkCollision(PhysicsObject* pOther) = 0;

    glm::vec2 getPosition() { return m_position; }
    float getRotation() { return m_rotation; }
    glm::vec2 getVelocity() { return m_velocity; }
    float getMass() { return m_mass; }

protected:
    glm::vec2 m_position;
    glm::vec2 m_velocity;
    float m_mass;
    float m_rotation;  //2D so we only need a single float to represent our rotation
};
```

Start by implementing the constructors, which should initialize the member variables.

The *applyForce()* function is where we apply Newton's second law. This function will modify the velocity of the object according to the object's mass and how much force has been applied.

Applying a force to an object will either speed it up or slow it down. That is to say, applying a force will change the *acceleration* of an object. By adding the acceleration to the current velocity, we find the object's new velocity.

As you would recall from the lecture, according to Newton's Second Law acceleration is calculated by dividing force by mass (a = F/m). So, in the *applyForce()* function you will need to calculate acceleration and add it to the *m_velocity* member variable. You should be able to write this as one line of code.

The *applyForceToActor()* function prototype is a variation of the *applyforce()* function. It allows us to simulate one actor "pushing" another.

You may recall Newton's third law, which states "for every action, there is an equal and opposite reaction". In the *applyForceToActor()* function you must first apply the input force to the input actor (by calling the *applyForce()* function of that actor), and then apply the opposite force to the current actor (by calling *applyForce()* on *this* using the negative force).

The *checkCollision()* function is a pure abstract function, ensuring that a programmer can not make an instance of the *RigidBody* class (they may only create instances of sub-classes). Each sub-class implementation will need to override this function to implement the specific collision detection algorithm required for that shape.

Finally, the *fixedUpdate()* function will apply any forces to the actor (i.e., gravity) before calculating the final position.

In this engine we're passing the gravity as an argument each time we call the *Rigidbody's fixedUpdate()* function. The gravity can be set when the application launces, and will be either zero (for a top-down pool table simulation) or non-zero (for an Angry Birds style game). Apply it as a force, but multiply by the object's mass first so that it acts as a pure acceleration. We'll divide by mass again inside our *applyForce()* function as per Newton's Second Law.

After applying gravity as a force, we add the updated vector to the actor's position to get the new position of the actor.

```
void Rigidbody::fixedUpdate(glm::vec2 gravity, float timeStep)
{
    applyForce(gravity * m_mass * timeStep);
    m_position += m_velocity * timeStep;
}
```

You may have noticed a *debug()* function. This can be left blank, or filled with your own debugging code (like logging the class's member variables to the console) to help you debug problems in your application.

The Sphere Class:

The definition for the sphere class is as follows:

```
class Sphere : public Rigidbody
{
public:
    Sphere(glm::vec2 position, glm::vec2 velocity,
        float mass, float radius, glm::vec4 colour);
    ~Sphere();

    virtual void makeGizmo();
    virtual bool checkCollision(PhysicsObject* pOther);

    float getRadius() { return m_radius; }
    glm::vec4 getColour() { return m_colour; }

protected:
    float m_radius;
    glm::vec4 m_colour;
};
```

We have a single constructor which allows us to instantiate an actor in our scene with a starting position, velocity, mass radius and colour. Make sure you call the base class (*Rigidbody*) constructor, passing in the appropriate shape type. This is the constructor definition for the *Sphere* class, other shape classes will be similar:

```
Sphere::Sphere(glm::vec2 position, glm::vec2 velocity, float mass, float radius,
glm::vec4 colour) :
    Rigidbody(SPHERE, position, velocity, 0, mass)
{
    m_radius = radius;
    m_colour = colour;
}
```

In the definition of the class, note how we have not added the *fixedUpdate(), applyForce()* or *applyForceToActor()* functions. These are implemented in the rigid body class, and as they will always be the same for all sub-classes, we won't need to overload them here. However, because all our shape sub-classes require a different type of gizmo to represent them visually, it makes sense to provide them with unique *makeGizmo()* functions.

The *makeGizmo* function will simply call the *add2DCircle()* function of the *Gizmo* class, passing in the appropriate arguments. This is a static function, and don't forget that the *Gizmo* class is defined within the *aie* namespace.

As the *checkCollision* function is defined as an abstract function in the *Rigidbody class*, we will need to implement this function for the *Sphere* class now.

This function takes another *PhysicsObject* as an argument, and should determine whether this sphere as collided with the other object. We will go into collision detection in much more detail in a future tutorial, so for now we are going to assume that the other object is always another *Sphere* (and if it isn't, we'll ignore it).

First, dynamically cast the *pOther* pointer to a *Sphere* pointer. If the cast succeeded, then find the distance between the centre points of the two spheres (this is simply one position subtracted from the other, although the *glm::distance()* also exists for this purpose). If this distance is less than the combined radius of both spheres, then the two spheres have collided. Return *true* if you detect a collision, otherwise *false*.

For now, you don't need to derive classes for the plane or the boxes. All our experiments with Newton's laws of motion and the projectile physics will be done with spheres.

The PhysicsScene Class:

We want to update our *PhysicsScene* class to check for collisions between all the spheres in our simulation.

Ideally, we'd want to implement some sort of special partitioning system to help optimize the number of collision checks we need to perform. However, for our purposes, a simple nested *for* loop will suffice (although you definitely wouldn't want to use this in any sort of real application).

Add the following code to your *PhysicsScene::update()* function:

```
void PhysicsScene::update(float dt) {
    static std::list<PhysicsObject*> dirty;

    // update physics at a fixed time step
    static float accumulatedTime = 0.0f;
    accumulatedTime += dt;

    while (accumulatedTime >= m_timeStep) {
        for (auto pActor : m_actors) {
            pActor->fixedUpdate(m_gravity, m_timeStep);
        }
        accumulatedTime -= m_timeStep;

        // check for collisions (ideally you'd want to have some sort of
        // scene management in place)
        for (auto pActor : m_actors) {
            for (auto pOther : m_actors) {
                if (pActor == pOther)
                    continue;
                if (std::find(dirty.begin(), dirty.end(), pActor) != dirty.end() &&
                    std::find(dirty.begin(), dirty.end(), pOther) != dirty.end())
                    continue;

                Rigidbody* pRigid = dynamic_cast<Rigidbody*>(pActor);
                if (pRigid->checkCollision(pOther) == true) {
                    pRigid->applyForceToActor(
                        dynamic_cast<Rigidbody*>(pOther),
                        pRigid->getVelocity() * pRigid->getMass());
                    dirty.push_back(pRigid);
                    dirty.push_back(pOther);
                }
            }
        }
        dirty.clear();
    }
}
```

Since we're using a nested *for* loop here, we want to avoid the case where we process a collusion using objects *A* and *B*, and then process that same collision using objects *B* and *A*. To avoid this I've made use of a *dirty* list. Any time a collision is found, both objects involved in that collision are added to the *dirty* list. This list can then be checked in future iterations of the loop so that we skip checking collisions on any object that has already been processed.

This is of course also not ideal. It will fall over in the situation where an object is involved with many collisions at once. But for our purposes it will suffice.

We can also extend our *PhysicsScene* class from last tutorial by adding a *debugScene()* function that will simply call the *debug()* function of each actor:

```
void PhysicsScene::debugScene()
{
    int count = 0;
    for (auto pActor : m_actors) {
        cout << count << " : ";
        pActor->debug();
        count++;
    }
}
```

And finally, now that we'll be adding actors to our *m_actors* vector, we'll want to ensure that we're deallocating our memory correctly when cleaning up our scene. Ensure you are deleting any objects in the scene in the scene destructor:

```
PhysicsScene::~PhysicsScene()
{
    for (auto pActor : m_actors)
    {
        delete pActor;
    }
}
```

Demonstrating Newton's First and Second Laws:

We'll start by demonstrating Newton's first law – an object in motion will remain in motion until acted upon by an external force.

To do this we'll turn off gravity, add a single sphere to our scene, and give our sphere an initial velocity. The sphere should remain travelling at a constant speed in a fixed direction.

Update the *startup()* function of your application class as follows:

```
bool Physics01_LinearForceApp::startup() {
    // increase the 2d line count to maximize the number of objects we can draw
    aie::Gizmos::create(255U, 255U, 65535U, 65535U);

    m_2dRenderer = new aie::Renderer2D();

    m_font = new aie::Font("./font/consolas.ttf", 32);

    // initialize the physics scene
    m_physicsScene = new PhysicsScene();
    m_physicsScene->setGravity(vec2(0, 0));
    m_physicsScene->setTimeStep(0.01f);

    Sphere* ball;
    ball = new Sphere(vec2(-40, 0), vec2(10, 30), 3.0f, 1, vec4(1, 0, 0, 1));
    m_physicsScene->addActor(ball);

    return true;
}
```

Compile and execute your program to see Newton's first law at work.

To simulate Newton's second law, simply set the gravity to a non-zero value, for example (0,-10). In this case the actor should move in a parabolic path simulating the movement of a projectile. Applying a force to the actor whilst the simulation is running further demonstrates the second law.

Demonstrating Newton's Third Law:

Newton's third law is trickier to demonstrate. For this, we will use the *Rigidbody::applyForceToActor()* function.

To demonstrate Newton's third law, instantiate two spheres of the same mass and radius next to each other in the centre of the screen.

In the application's *update()* function scan for a key press. When the key is pressed use the *applyForceToActor()* to apply a force, once, to one of the actors. Alternatively, call the *applyForceToActor()* function at the end of the *startup()* function, leaving the *update()* function unmodified.

The force you apply must be towards the second actor. You should find that the two actors move away from each other with equal velocities.

Repeat the experiment but try it with spheres of different masses.

```
    m_physicsScene->setGravity(vec2(0, 0));  // turn off gravity

    Sphere* ball1 = new Sphere(vec2(-4, 0), vec2(0, 0), 4.0f, 4, vec4(1, 0, 0, 1));
    Sphere* ball2 = new Sphere(vec2(4, 0), vec2(0, 0), 4.0f, 4, vec4(0, 1, 0, 1));

    m_physicsScene->addActor(ball1);
    m_physicsScene->addActor(ball2);

    ball1->applyForceToActor(ball2, vec2(2, 0));
```

Simulating a Collision:

For this simulation, create two spheres some distance apart on the screen. Give each sphere an initial velocity such that it moves towards the other sphere. After some time the spheres will collide (the collision will be detected by the placeholder collision detection and response logic we placed inside the *PhysicsScene* class). After the spheres collide, you should see them bounce off each other and travel in the opposite direction.

```
// turn off gravity
m_physicsScene->setGravity(vec2(0, 0));

Sphere* ball1 = new Sphere(vec2(-20, 0), 0, 0, 4.0f, 4, vec4(1, 0, 0, 1));
Sphere* ball2 = new Sphere(vec2(10, 0), 0, 0, 4.0f, 4, vec4(0, 1, 0, 1));

m_physicsScene->addActor(ball1);
m_physicsScene->addActor(ball2);

ball1->applyForce(vec2(30, 0));
ball2->applyForce(vec2(-15, 0));
```

Simulating a Rocket Motor:

Finally, you are going to simulate a simple rocket motor. Create a sphere in the centre of the screen, this is the rocket. Set gravity to zero. In the update function, at time intervals, you need to:

- Reduce the mass of the rocket by *M* to simulate fuel being used
- Create a new sphere of mass *M* next to the rocket to simulate an exhaust gas particle (ensuring that the two spheres won't collide – in fact, this simulation will work better if you turn off collision detection altogether)
- Use *applyForceToActor()* to apply force to the exhaust gas from the rocket (make sure it is in the correct direction)
- Repeat until all the mass has been used up

You will need to experiment with different forces and masses, firing rate, to make it work properly.

For added effect make the exhaust gas particles smaller than the rocket and a different colour. Try turning gravity on and see if you can get the rocket to lift off against gravity. Try changing the direction of the force you apply to steer the rocket around the screen.