Physics - Linear Motion

Introduction

Modern video games may have 100s of objects moving and interacting on screen in a believable manner. At the core of any game with realistic object movement is a *physics engine*, comprised of the functions and classes required to model the complex interactions between objects in a realistic, real-time manner. At its core, a physics engine must do two things: It must enable objects to move in a physically realistic manner as forces are enacted upon them, and it must detect when collisions occur between those objects and allow those objects to react accordingly.

In this tutorial, we'll take a look at the basics of enabling realistic object motion, and learn how to add in concepts such as acceleration, velocity, and mass into our game simulations.

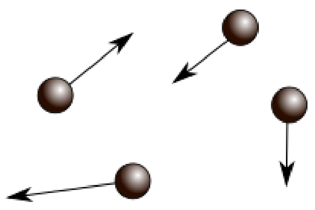
Physics Bodies

The scenes in which video games are built up of may contain many hundreds of objects, with many considerations to be made as to the level of accuracy required in their replication of the 'real world'. Broadly speaking, there are two factors that influence an object's physical properties - whether they can *cause* physical interactions, and whether they *react* to physical interactions. The walls in a typical FPS game can be collided with, preventing the player from passing through them like a ghost - the walls cause a collision to be triggered in the game code somewhere, which will at some point then be resolved (pushing the player back, and making them slide along it instead). However, to generally don't *react* to any interactions, except perhaps graphically - bumping into that wall doesn't impart any force upon the wall to push it slightly, and even if you shoot it and cause bullet holes to pockmark its surface, those are just graphical effects; the physical properties of the wall have not been changed.

In game engine terms, the example wall above may have a *collision volume* (like the AABBs and spheres we looked at to do raycasting) to detect that the player had intersected it, but lacks a *physics body*, which models how forces should be applied to an object, and how it should react when collisions occur. These physics bodies can be broadly categorised as one of three things - *particles*, *rigid bodies*, and *soft bodies*, each of which may be used in a game to simulation different types of object.

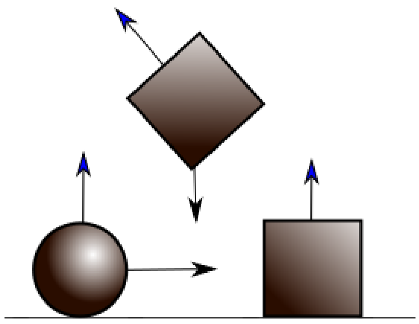
Particles

The simplest form of physical representation is the *particle*. A particle has a position in space, and can have velocity and accelerations applied to it. While a particle may have mass, it doesn't have an orientation; it's just a point in space representing the presence of 'something' . This can be seen as an extension of the 'particle effects' often used in video games to visually represent things like fire and smoke - each 'particle' is a little texture in world space that can move, but its orientation doesn't particularly matter to the overall simulation it is part of. Sometimes particle systems employ physics calculations to improve their realism - spark effects from a glowing fire may bounce off surfaces, and water particles being emitted from a water fountain may eventually be pulled down by the effect of gravity.



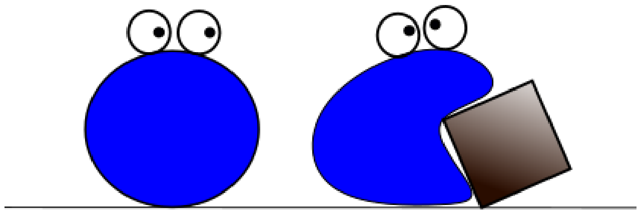
Rigid Bodies

A step up from a particle is a rigid body - rather than a set of particles building up a shape, this can be seen as a single body that has both volume, and an orientation (and as we'll see later, the ability to change its orientation via forces). We can assume rigid bodies to be the solid shapes that make up most objects in the world - we can determine the volume and orientation of a cup, just as easily as we can a spaceship. While the cup (or spaceship!) is really composed of multiple elements (the lid, the cup itself, and the cardboard sleeve), and beyond that multiple *atoms*, we can assume that the positions between them remain fixed, and therefore treat them all as part of a single whole. As forces interact with a rigid body, they do not change shape or change the distribution of their mass throughout their volume in any way, they are a fully 'fixed' shape.

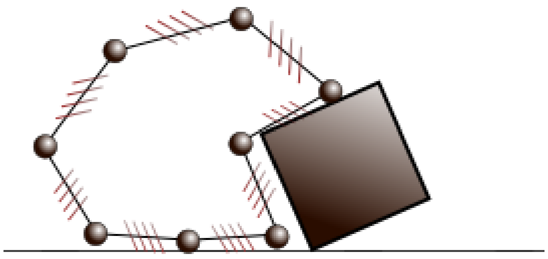


Soft Bodies

Most objects in our games can be assumed to be rigid bodies - they never change their shape, except perhaps via animations. Some objects require more physically accurate dynamics to fully model their movement in the world - think about a superhero's cape billowing in wind, or maybe a player character made of squishy putty that can stretch and squish according to player input:



To create effects like this, we need a *soft* body, rather than a rigid one. Instead of a single solid shape to represent the physical properties of our object, we can instead treat the object as a set of points, connected via springs:



As each of the points has forces applied to it (either by colliding with the world in some way, or direct application of force using the joypad controller), the springs stretch and contract, spreading that force out, and making the shape as a whole move or 'squish'. If we imagine those 'points' are the vertices of a mesh, it's easy to visualise how moving the points around in a physically consistent way would create a flag waving effect, or a stretchy/squishy effect. However, getting those points to interact correctly with the world around it is much harder than with rigid bodies - instead of a simple shape like a cube or sphere (both of which have easy methods of determining whether something is touching them, as we'll see later in the tutorial), soft bodies require many checks to see which points are being pushed around by objects in the world. The effect of the springs is tricky, too; while the forces applied on one point should interact with the connected points in a consistent manner, small amount of floating point errors can cause soft bodies to 'jiggle' around even when they should be at rest, and if enough of this jiggling energy accumulates, the soft body can deform, or even break entirely.

Physics Terms

In this tutorial series, we'll be encountering a number of terms which you may or may not have come across. Just in case, we'll go over some of these terms here, to resolve any ambiguity.

Displacement

In the graphics module, we became familiar with changing the world-space position of the objects in the world. In some literature relating to physics calculations, you will see position sometimes referred to as *displacement*, where it specifically relates to how much the object has moved from its initial position over time, and is usually stored as a vector, with the displacement on each axis stored using meters as its standard unit of measurement. In any calculations relating to deriving values, we'll be using p to denote position; in classical mechanics, the actual displacement vector (or our object's position vector, as we're used to thinking of it from the previous module) is additionally denoted s, from the latin *spatium*.

Velocity

The first derivative of the object's displacement with respect to time (that is, how much the position of the object changes over time) is known as its *velocity*. As with displacement, it is a 3D vector - our objects move in a particular direction (the *direction* of the velocity vector), at a given *speed* (the *magnitude* of the velocity vector). Changes in the velocity vector of a physics body mean that it is either slowing down (the magnitude is getting closer to zero), or speeding up (the magnitude is getting larger). We would usually describe the magnitude of an object's velocity using meters per second as the measurement unit - after 1 second, an object with a velocity of x, will have moved in the world by $\|x\|$ meters, in the direction of $\hat{x}$; you will sometimes also see this stated as being a velocity of $x\ m/s$ or ms^{-1}. In physics calculations, an object's velocity is often denoted $\dot{p}$ when discussing changes in an object's position generally, or sometimes just v.

Acceleration

The second derivative of a body's position with respect to time is known as it's acceleration (or to put it another way, acceleration measures the *rate of change* of velocity). In physics calculations, its often denoted as $\ddot{p}$, or less commonly $\dot{v}$, when reasoning specifically about rates of change, and often just a when it appears in other calculations. In a car, we push down on the accelerator pedal to speed up (within legal speeding limits, of course!), by causing the engine to do more work; this work changes the car's velocity, and therefore its position over time. Acceleration is usually measured in meters per second *per second*, often denoted as m/s^2, or alternatively ms^{-2}.

The most popular example of acceleration is probably the Earth's gravity, which you probably know as 9.8 m/s^2. This means that physics bodies move downwards at an increasing rate of 9.8 metres per second *per second* - that is, every second, the rate of change of velocity will have increased by 9.8 metres per second, so the body speeds up, and thus changes position faster and faster (until it reaches its *terminal velocity* - when the 'drag' slowdown caused by hitting the air equals the acceleration force, cancelling it out).

The difference between acceleration and velocity is important - moving forward at a constant rate with a velocity of 9.8 m/s is pretty fast, and you'd complete a 100 metre sprint in just over 10 seconds; *accelerating* at a constant rate of 9.8 m/s^2 will mean that after 10 seconds you're moving at *98 metres per second*, and will get you a gold medal at the Olympics.

Force

To move, an object must have a *force* applied to it. This force will have a direction, and a magnitude, and so can be represented in code as a vector. The unit of measurement of force is the *Newton*, with one Newton being the force required to move a 1kg weight at a rate of 1 metre per second *per second*. Therefore, a force is an adjustment to a rigid body's acceleration.

Mass

Mass is the measurement of how much matter an object is made out of, and is measured in kilograms. Matter resists forces, so the more mass an object has, the harder it is to move it. This isn't the same as how much that object weighs, although the two are related. Weight is how much force an object applies under gravity - a 1kg mass applies 9.8 *Newtons* of force on Earth due to the planet's gravity, but the same object applies less force on the Moon as there's less gravity to pull it down - so it 'weighs less'...even though it's still 1kg of mass. That's why although things *weigh* nothing in space, they still require big rocket motors - to counteract the *mass* of the object.

In physics engines, every object we wish to move and interact in our simulation will have a mass value, measured in grams or kilograms (or some value derived from this). Generally, we store the *reciprocal* of mass - that is, $\frac{1}{mass}$. This 'inverse' mass value helps turn some division operations into multiplication operations, and also provides some useful side effects that we'll see later.

Momentum

Another term you may come across in physics engine literature is *momentum*. This is simply the product of an object's mass, and velocity:

$$\mathbf{p} = m\mathbf{v}$$

Momentum is measured in kg m/s - simply the units of measurement of both parts of the momentum calculation. You may have heard of this in terms of the *conservation* of momentum. What this refers to is that the total momentum of a system of physics bodies should always be constant - as objects collide, they impart some of their momentum on the other object, reducing their own. This is why when the cue ball hits another ball in snooker the cue ball may stop, but the other ball then speeds up; the total amount of momentum in the system has been conserved, even if the sum of velocities has not (the cue ball could be heavier or lighter than the other balls, but it would still impart momentum such that the system as a whole remained constant).

Newton's Laws of Motion

The fundamentals of motion on objects can be concisely described using Isaac newton's three laws of motion - everything we later do with moving objects and reacting to collisions between objects relates to them, so having some understanding of them will be useful as we continue through the tutorial series.

Newton's First Law

The first law states that an object either stays at rest, or moves at constant velocity, unless acted upon by some other force. For our physics engine, this means that an object should, by default, not do very much! It shouldn't move until something either hits it, or the object itself exerts some force to move it in a direction. Once the object *does* start moving, it should keep moving until some other force slows it down or changes its direction. Think of a spaceship, floating out in deep space - there's nothing pushing it around or otherwise enacting forces on it (except an *incredibly* small gravity force from nearby astral bodies, which we'll just ignore for now!), so it can remain still. If the spaceship fires up its rocket engines for a bit, it'll start to move. It will then keep moving in that direction, even if the engines are turned off again, as there's nothing acting on it to slow it down. Even rotating the ship around won't change the direction or speed of the movement - until the engines are fired up again, anyway.

Inertia

Newton's first law is sometimes known as the *law of inertia*. Inertia is an object's resistance to a change in its velocity, and is related to its mass - just as it is harder to move a heavier object, so is it harder to change the direction of an object once it does start moving.

Friction & damping

On Earth, you've probably noticed while riding a bike or driving a car that if you stop pedaling or turn the engine off, you don't just keep going, but instead come to a stop - unless you're going downhill! Gravity enacts a force upon everything with mass, and will therefore pull objects downhill, but on a level surface, we'd come to a stop due to *friction* - the tires of the bicycle or car scrape against the road, adding some force, and the chassis of the car/bicycle also gets slowed by air resistance - the friction force caused by trying to move through the air. Calculating the amount of friction (from either tire or air) upon an object is a computationally tricky problem, so we generally don't do it in a real-time physics engine for games. Instead we model friction by slightly *damping* velocity every update, by multiplying it by a scalar value - as long as this value is slightly less than 1.0, it will cause objects to slowly lose their velocity, as if being slowed by friction.

Newton's Second Law

The second law states that the sum of forces acting upon an object is equal to that object's mass multiplied by the acceleration of the object, or:

$$\mathbf{F} = ma$$

In a physics engine, we generally apply forces to objects in code, rather than directly applying an acceleration. We might say that the character jumping exerts a force of 1000 *Newtons* (the unit for the measurement of force comes from this law!), for instance. This force would then be integrated into the velocity and position of our player object over time. Therefore, it's better rearrange the equation like this:

$$a = \frac{\mathbf{F}}{m}$$

In this way, we can think of working out the new acceleration of an object as being the sum of all forces applied, divided by mass. Remember earlier when it was mentioned that in physics engines we generally deal with *inverse* mass? This equation is a good explanation as to why. Rather than dividing by m, we can instead multiply by the *inverse* of mass:

$$a = \mathbf{F}m^{-1}$$

Division is generally a slower operation than multiplication (that is, it takes more CPU cycles), so by doing this, we speed up our physics calculations. But there's another benefit! As was mentioned earlier, inverse mass allows for an easier time adding in objects that we don't want to move in our games, simply by setting an inverse mass of 0.0:

$$a = 0 = \mathbf{F} \cdot 0$$

In this way, no matter how much force is applied to the object (from things colliding with it etc), it will not move at all, as the resulting acceleration on it will also end up being zero. This is useful for things like the 'floor' of the level, where we want the player to be able to stand on and jump on the object (requiring some physics interactions to solve any collision detection), but we really don't want the floor to move - imagine if the levels in Mario were slightly pushed downwards every time Mario jumped!

Newton's Third Law

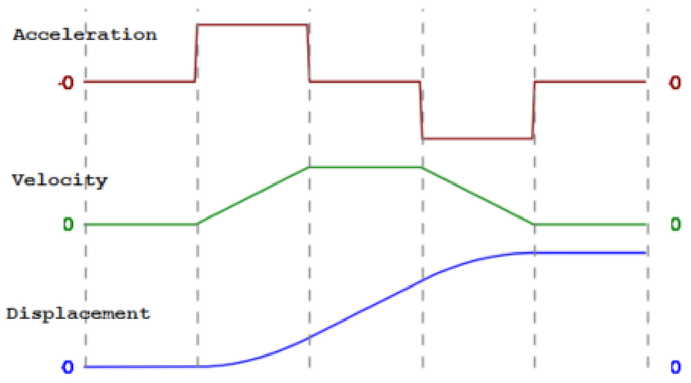
In the Mario example above, you might be wondering why the floor would move whenever Mario jumped, if it were not for the 'infinite' mass afforded by using inverse mass in our calculations. This is due to newton's third law of motion, which states that when an object exerts force on second object, the second object simultaneously exerts an equal force in the opposite direction; this is more commonly stated as - *for every action, there is an equal and opposite reaction.*

Generally speaking, this means in our physics engine that whenever a collision occurs, both objects end up receiving forces - in a game of Snooker, if the cue ball hits another ball, the second ball receives some force enacted upon it by the cue ball, but the cue ball also slows down as an equal and opposite force is applied to it via the second ball.

In our Mario example, this means that as he jumps, really there should be an opposite force applied to the world. In reality, the jump interaction would *probably* be coded to just apply the upwards force directly to Mario and skip the interaction - but now think about when he lands again; that's a collision that must be detected and resolved, and inverse mass ensures we don't push the world around when we land.

Numerical Integration

We know that the change in an object's position over time is described by the velocity property, and the rate of change of velocity as acceleration. To actual perform these changes we need to do some more calculus, and do some *integration.* As you may well already know, to integrate a function (the *integrand*), we evaluate the result of the function at smaller and smaller intervals to get a progressively more accurate idea of the area under the curve on the plot of the function:



When describing the rate of change of velocity and acceleration, we often see the following equations:

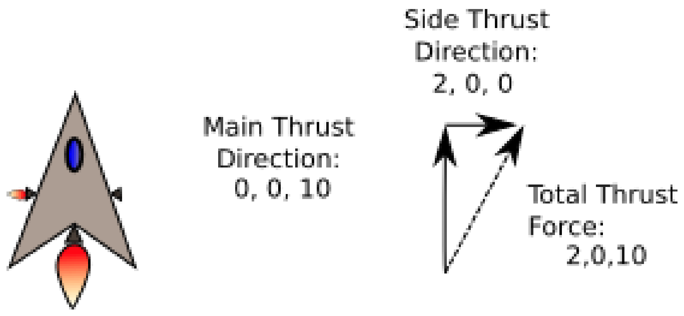
$$v = \frac{dp}{dt}$$
$$a = \frac{dv}{dt}$$

When describing the *integration* of velocity and acceleration, we would instead use these equations:

$$v = \int a dt$$
$$p = \int v dt$$

In each case, dt is the change in time. As we're dealing with a real-time video game simulation comprised of a discrete series of frames that are rendered sequentially, this change in time is reasonably obvious - 1 second divided by the framerate will give us the dt for each frame.

In a physics engine, we usually determine the total amount of force applied to an object that frame, by summing all of the individual forces that may be affecting it - a spaceship that is both firing its main engines to move forward, and its side thruster to move to the right, will have a total force applied that should make it move diagonally:



From there, we can determine the amount of acceleration the spaceship is undertaking using the previously mentioned equation $a = Fm^{-1}$. To actually then determine the spaceship's new position, the acceleration value should be *integrated*, to determine the change in amount of the spaceship's velocity, and from there integrate velocity to determine the change in the ship's position in the current frame.

There's a problem, though. If the ship was accelerating at $10ms^{-2}$ after 1 second, and $25ms^{-2}$ after 2 seconds, how fast was the object accelerating at 1.5 seconds? We dont really know from just those time points whether acceleration was increasing linearly or not, so treating these as discrete changes can lead to inaccuracy compared to 'real life' conditions. This makes the number of times we update the physics system per second *very* important - the more we slice up each second into discrete frames (and then integrate by that frame's dt), the more accurate the physics system is able to model the changes in acceleration and velocity, and the closer to the 'real' answer we can get.

There's a number of numerical integration methods commonly used in physics engines to determine the new position of the physics objects in the simulation each frame, each of which has different features and drawbacks.

Explicit Euler

Explicit Euler integration (sometimes simply 'Euler Integration') is the most simple of the discussed integration methods. Every simulation update, we determine the follow new values:

$$v_{n+1} = v_n + a_n dt$$
$$p_{n+1} = p_n + v_n dt$$

This states that for our next frame's values (denoted as *n+1*) are formed by taking the current frame's values (denoted n), and add the derivative, multiplied by the timestep. We implicitly know what the acceleration of the current frame will be, as any forces applied in the frame can be summed, and multiplied by the inverse mass to obtain it, as we saw earlier.

Implicit Euler

Explicit Euler integration isn't perfect - velocity should be changing throughout the timestep due to the integration of acceleration, but we're treating it simply as a constant addition to change the position. This approximation leads to inaccuracy over time, for any object with a changing amount of velocity over time. A more complete integration would instead compute the derivatives at the *next* time step, like so:

$$v_{n+1} = v_n + a_{n+1} dt$$
$$p_{n+1} = p_n + v_{n+1} dt$$

This is known as *implicit Euler*, or sometimes 'Backward Euler' integration. This is fine for integrating data from a complete data set, as the 'next' velocity and acceleration would be available to integrate. This isn't really any good for a real time simulation, unless we know exactly what acceleration will be applied in the next frame (maybe it's constantly increasing by a fixed rate, or some other simple function), as our gameplay interactions could result in unique combinations of collisions and forces each frame. We can try and *predict* the derivatives by trying to fit them to curves, but the predictions may be incorrect, resulting in inaccuracy, which we're trying to avoid!

Semi - Implicit Euler

There's a middle ground between implicit and explicit Euler integration, known as *semi-implicit Euler* (or also sometimes *Symplectic Euler*) integration. In this case, we integrate our second derivative (acceleration) using the current state, allowing us to then integrate our first derivative (velocity) with a more up to date state:

$$v_{n+1} = v_n + a_n dt$$
$$p_{n+1} = p_n + v_{n+1} dt$$

In practice, this is as simple as flipping around the order in which we would calculate the new velocity and position of an object, making it no slower than explicit Euler to calculate, but more accurate, and therefore less likely to result in problems over time.

Verlet

If we know the current position of an object, the previous position, and the timestep between those two measurements, we can determine the velocity directly, without having to store it separately. This is the basis for the *Verlet* integration method:

$$p_{n+1} = p_n + (p_n - p_{n-1}) + a_n dt^2$$

This method is sometimes used for particle systems computed on the GPU, as the cost of reconstructing velocity can be smaller than the impact of reading from another buffer holding the velocity data, and the resulting cache misses that reading from that buffer will cause. The downside to Verlet integration is that we have to do some additional calculations if we want to have an object that is already moving at the start of the simulation - if there's no previous position (or worse, the previous position variable is defaulted to the origin) then we get a wholly inaccurate predicted velocity for the object.

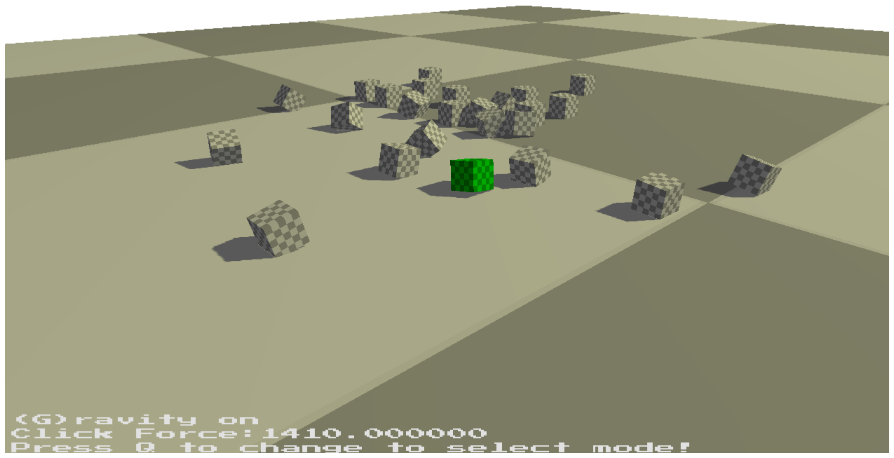
Runge-Kutta methods

We can take integration further by splitting up the timestep, and performing integration multiple times, to try and get a greater understanding of how velocity and position changes over the course of the timestep. One such prediction method is the *Runge Kutta* method, which takes an average of multiple splits of the frame's timestep. Different numbers of steps can be taken, leading to slightly different results - taking 2 steps is known as 'RK2', and 4 as 'RK4'.

Using this allows for greater accuracy in position over time, but comes at the cost of each objects integration now taking 4 times as much computation. In cases where total accuracy is not required, it's often better to spend that computation time moving the simulation forward, and working out new collisions and movements that way - higher framerates are often better for the gameplay experience than more 'accurate' individual frames.

Tutorial Code

We now know how position, velocity, and acceleration are related, and how forces applied to objects interact with mass to result in accelerations upon an object. To demonstrate this, we're going to expand on the test program created in the previous tutorial to implement some simple integration of linear motion on some basic rigid bodies. As part of this, we're going to expand upon the **PhysicsObject** class to allow implement *rigid body* dynamics, with a semi-implicit Euler integrator, to allow us to see how most games handle their physics calculations.
To demonstrate the rigid body dynamics and linear motion, we're going to expand upon the previous tutorial, and not just be able to click on an object, but then push it around as well, by applying forces to it.



```
(G)ravity on
Click Force:1410.000000
Press Q to change to select mode!
```

To do this, we need to fill in another couple of currently empty methods, *IntegrateAccel* and *IntegrateVelocity*, which are both in the **PhysicsSystem** class. To see what functionality these new methods are going to require, let's take another look at the **PhysicsObject** class, and the variables it holds related to linear movement:

PhysicsObject Class

```
public:
      void ClearForces();
      void AddForce(const Vector3& force);
      Vector3 GetLinearVelocity() const {return linearVelocity;}
protected:
      const CollisionVolume*  volume;
      Transform*              transform;
//We'll be using these variables!
      float    inverseMass;
      Vector3  linearVelocity;
      Vector3  force;
```

PhysicsObject class header

Applying a force is easy - we have a single method, which takes in a **Vector3** representing a direction and an amount of Newtons of force to apply in that direction, and adds it to our new force variable. An object might have multiple forces applied to it in a single frame, so make sure you are adding, not setting this value!

```cpp
void PhysicsObject::AddForce(const Vector3& addedForce) {
    force += addedForce;
}
```

PhysicsObject::AddForce method

At the end of every frame, we're going to zero out any forces, so that they are only applied once, by using the *ClearForces* method:

```cpp
void PhysicsObject::ClearForces() {
    force     = Vector3();
}
```

PhysicsObject::AddForce method

This means that any forces applied are *instantaneous* applications of n Newtons, rather than n Newtons over time.

PhysicsSystem Class

In the **PhysicsSystem** class, we need to add code to two methods. The first is *IntegrateAccel*, which, for a given timestep *dt* determines the correct amount of acceleration to apply for a given set of forces applied each frame. Add in the following code to the currently empty method:

```cpp
 1 void PhysicsSystem::IntegrateAccel(float dt) {
 2     std::vector<GameObject*>::const_iterator first;
 3     std::vector<GameObject*>::const_iterator last;
 4     gameWorld.GetObjectIterators(first, last);
 5
 6     for (auto i = first; i != last; ++i) {
 7         PhysicsObject* object = (*i)->GetPhysicsObject();
 8         if (object == nullptr) {
 9             continue; //No physics object for this GameObject!
10         }
11         float inverseMass = object->GetInverseMass();
12
13         Vector3 linearVel = object->GetLinearVelocity();
14         Vector3 force     = object->GetForce();
15         Vector3 accel     = force * inverseMass;
16
17         if (applyGravity && inverseMass > 0) {
18             accel += gravity; //don't move infinitely heavy things
19         }
20
21         linearVel += accel     * dt; //integrate accel!
22         object->SetLinearVelocity(linearVel);
23     }
24 }
```

PhysicsSystem::IntegrateAccel method

This method operates on each **GameObject** in the game world in turn, and so uses a couple of iterators, and then iterates through each **GameObject**, and if it has a **PhysicsObject**, calculates the amount of acceleration as simply $f \cdot m^{-1}$ (line 15), and from it, integrates a new velocity using the timestep (line 21). On line 17, we're also going to apply some gravity; this force doesn't get affected by an object's mass, *unless* we're assuming it is infinitely massful, and so should *never* move.

Integrating the resulting velocity into an object's position is much the same process as integrating acceleration. In the *IntegrateVelocity* method, we need to add the following code:

```
 1 void PhysicsSystem::IntegrateVelocity(float dt) {
 2    std::vector<GameObject*>::const_iterator first;
 3    std::vector<GameObject*>::const_iterator last;
 4    gameWorld.GetObjectIterators(first, last);
 5    float frameLinearDamping  = 1.0f - (0.4f * dt);
 6
 7    for (auto i = first; i != last; ++i) {
 8       PhysicsObject* object = (*i)->GetPhysicsObject();
 9       if (object == nullptr) {
10          continue;
11       }
12       Transform&transform = (*i)->GetTransform();
13       //Position Stuff
14       Vector3 position  = transform.GetPosition();
15       Vector3 linearVel = object->GetLinearVelocity();
16       position += linearVel * dt;
17       transform.SetPosition(position);
18       // Linear Damping
19       linearVel = linearVel * frameLinearDamping;
20       object->SetLinearVelocity(linearVel);
21    }
22 }
```

PhysicsSystem::IntegrateVelocity method

It's almost exactly the same process, except this time we're getting velocity and integrating it by *dt* (line 17). We're also going to simulate a small amount of drag or air resistance per frame, by slightly scaling down the linear velocity by an amount based on the framerate, such that we *should* get a reasonably consistent amount of velocity reduction no matter the framerate.

That's all we have to add to the codebase, as the *IntegrateAccel* and *IntegrateVelocity* methods are already being called by the *PhysicsSystem::Update* method, it's just that before, the methods were empty! Along with these, it also calls a *ClearForces* method, which goes through every **PhysicsObject**, and calls *ClearForces* on it, ready for the next frame's worth of gameplay code.

TutorialGame Class changes

To finish off our addition of linear motion into our physics engine, we're going to fill in another method, this time in the **TutorialGame** class. The *MoveSelectedObject* method is called every frame in the *UpdateGame* method, and will use the raycasting functionality we filled in in the previous tutorial to once again detect which object is being clicked - if it's the same object as was 'selected' previously, we'll push the **PhysicsObject** in the direction of the ray, scaled by a selectable number of Newtons, based on how much the scroll wheel has been pressed (we additionally output this variable to the screen on line 25 so its easier to tell how much an object will be pushed by).

```
23 void TutorialGame::MoveSelectedObject() {
24    renderer->DrawString("Click Force:" + std::to_string(forceMagnitude),
25       Vector2(10, 20)); //Draw debug text at 10,20
26    forceMagnitude += Window::GetMouse()->GetWheelMovement() * 100.0f;
27
28    if (!selectionObject) {
29       return;//we haven't selected anything!
30    }
```

PhysicsSystem::IntegrateVelocity method

```
    //Push the selected object!
    if (Window::GetMouse()->ButtonPressed(NCL::MouseButtons::RIGHT)) {
       Ray ray = CollisionDetection::BuildRayFromMouse(
                             *world->GetMainCamera());
       RayCollision closestCollision;
       if (world->Raycast(ray, closestCollision, true)) {
          if (closestCollision.node == selectionObject) {
             selectionObject->GetPhysicsObject()->
                AddForce(ray.GetDirection() * forceMagnitude);
          }
       }
    }
}
```

PhysicsSystem::IntegrateVelocity method

Conclusion

After making these changes, we should have some new functionality in our game code. Once an object has been selected, we can switch back to camera movement mode, and then click on the selected object, where it should (hopefully!) be pushed around in the direction of the mouse. Even more, we should be able to press the **G** key to toggle gravity, upon which all of the objects that spawn floating in the air will fall to the floor, while the floor stays still (due to being infinitely massive in our physics simulation). So far though, there's nothing to detect when the objects have actually hit the floor, so they'll appear to simply move straight through the floor!

Pressing **Q** will still swap between object selection mode and camera movement mode, and along with being able to click on objects with the left mouse button, the right mouse button will apply a force to it, the magnitude of which can be adjusted using the mouse wheel.

In this tutorial, we've seen the basics of how to implement a basic **rigid body** into our game world, that can move around using the **integration** of **acceleration** and **velocity** into its world **position**. We've seen how important the **frame time** is for proper integration and stability of our physics simulation, and how **mass** and **damping** interacts with the movement of our objects over time. While playing around with the project, you've probably noticed that things don't feel quite right - there's no collisions for one thing, but also no rotation of any sort. As the tutorial series progresses, we'll be tackling these problems to produce a more complete physics simulation starting with the addition of angular motion to our integration method, allowing objects to spin as forces are applied.

Further Work

1) The gravity value for the physics engine can be set via the *SetGravity* method - try changing this value to see how it affects the game objects.

2) At the moment the damping factor for linear velocity is hard coded to 0.4. Try adding a new *linearDamping* variable to the **PhysicsSystem**, and add some getters and setters to see how damping affects how far objects seem to slide aftr being 'pushed' by the mouse.

3) Try adding in keys that will modify the position of the selected object using forces. The *MoveSelectedObject* method is a good place to add this extra functionality.

4) Check out `https://github.com/NVIDIAGameWorks/PhysX-3.4/blob/master/PhysX_3.4/Source/LowLevelDynamics/src/DyBodyCoreIntegrator.h` to see Nvidia's implementation of linear motion in its PhysX engine, including damping.