

Physics in 3D

How to simulate the motion of rigid bodies

Posted by Glenn Fiedler (<http://web.archive.org/web/20181107181511/https://gafferongames.com/about>) on Thursday, September 2, 2004

Introduction

Hi, I'm Glenn Fiedler (<http://web.archive.org/web/20181107181511/https://gafferongames.com/about>) and welcome to **Game Physics** (<http://web.archive.org/web/20181107181511/https://gafferongames.com/categories/game-physics/>).

In the [previous article](http://web.archive.org/web/20181107181511/https://gafferongames.com/post/fix_your_timestep/) (http://web.archive.org/web/20181107181511/https://gafferongames.com/post/fix_your_timestep/) we discussed how to integrate our physics simulation forward at fixed delta time increments, regardless of display framerate.

In this article we are going to simulate motion in three dimensions.

Rigid Bodies

We will concentrate on a type of object called a **rigid body**. Rigid bodies cannot bend, compress or deform in any way. This makes their motion much easier to calculate.

To simulate the motion of rigid bodies, we must study both rigid body kinematics and rigid body dynamics. Kinematics is the study of how an object moves in the absence of forces, while dynamics describes how an object reacts to them. Together they provide all the information you need to simulate the motion of a rigid body in three dimensions.

Along the way I will show you how to integrate vector quantities, handle rotations in three dimensions and integrate to find the motion of your object as it moves and spins around the world.

Moving in the Third Dimension

As long as we only have single floating point values for position and velocity our physics simulation is limited to motion in a single dimension, and a point moving from side to side on the screen is pretty boring!

https://gafferongames.com/post/physics_in_3d/

Go

AUG NOV DEC

07

2017 2018 2019

▼ About this capture

5 captures

21 Oct 2017 - 7 Nov 2018

in turn to find the motion of the object in three dimensions.

Or... we could just use vectors.

Vectors are a mathematical type representing an array of numbers. A three dimensional vector has three components x, y and z. Each component corresponds to a dimension. In this article x is left and right, y is up and down, and z is forward and back.

In C++ we implement vectors using a struct as follows:

```
struct Vector
{
    float x,y,z;
};
```

Addition of two vectors is defined as adding each component together. Multiplying a vector by a floating point number is the same as just multiplying each component. Lets add overloaded operators to the vector struct so that we can perform these operations in code as if vectors are a native type:

```
struct Vector
{
    float x,y,z;

    Vector operator + ( const Vector &other )
    {
        Vector result;
        result.x = x + other.x;
        result.y = y + other.y;
        result.z = z + other.z;
        return result;
    }

    Vector operator*( float scalar )
    {
        Vector result;
        result.x = x * scalar;
        result.y = y * scalar;
        result.z = z * scalar;
        return result;
    }
};
```

Now instead of maintaining completely seperate equations of motion and integrating separately for x, y and z, we convert our position, velocity, acceleration and force to vector quantities, then integrate the vectors directly using the equations of motion from the [first article](http://web.archive.org/web/20181107181511/https://gafferongames.com/post/integration_basics/) (http://web.archive.org/web/20181107181511/https://gafferongames.com/post/integration_basics/):

```
F = ma
dv/dt = a
dx/dt = v
```

Notice how **F**, **a**, **v** and **x** are written in bold. This is the convention used to distinguish vector quantities from single value (scalar) quantities such as mass m and time t.

https://gafferongames.com/post/physics_in_3d/

Go

AUG NOV DEC

07

2017 2018 2019

About this capture

5 captures

21 Oct 2017 - 7 Nov 2018

operators for adding two vectors together, and multiplying a vector by a scalar, and this is all we need to be able to drop in vectors in place of floats and have everything just work.

For example, here is a simple Euler integration for vector position from velocity:

```
position = position + velocity * dt;
```

Notice how the overloaded operators make it look exactly the same as an Euler integration for a single value. But what is it really doing? Lets take a look at how we would implement vector integration without the overloaded operators:

```
position.x = position.x + velocity.x * dt;
position.y = position.y + velocity.y * dt;
position.z = position.z + velocity.z * dt;
```

As you can see, its exactly the same as if we integrated each component of the vector separately! This is the cool thing about vectors. Whether we integrate vectors directly, or integrate each component separately, we are doing exactly the same thing.

Structuring for RK4

In the example programs from previous articles we drove the simulation from acceleration assuming unit mass. This kept the code nice and simple, but from now on every object will have its own mass in kilograms so the simulation needs be driven by forces instead.

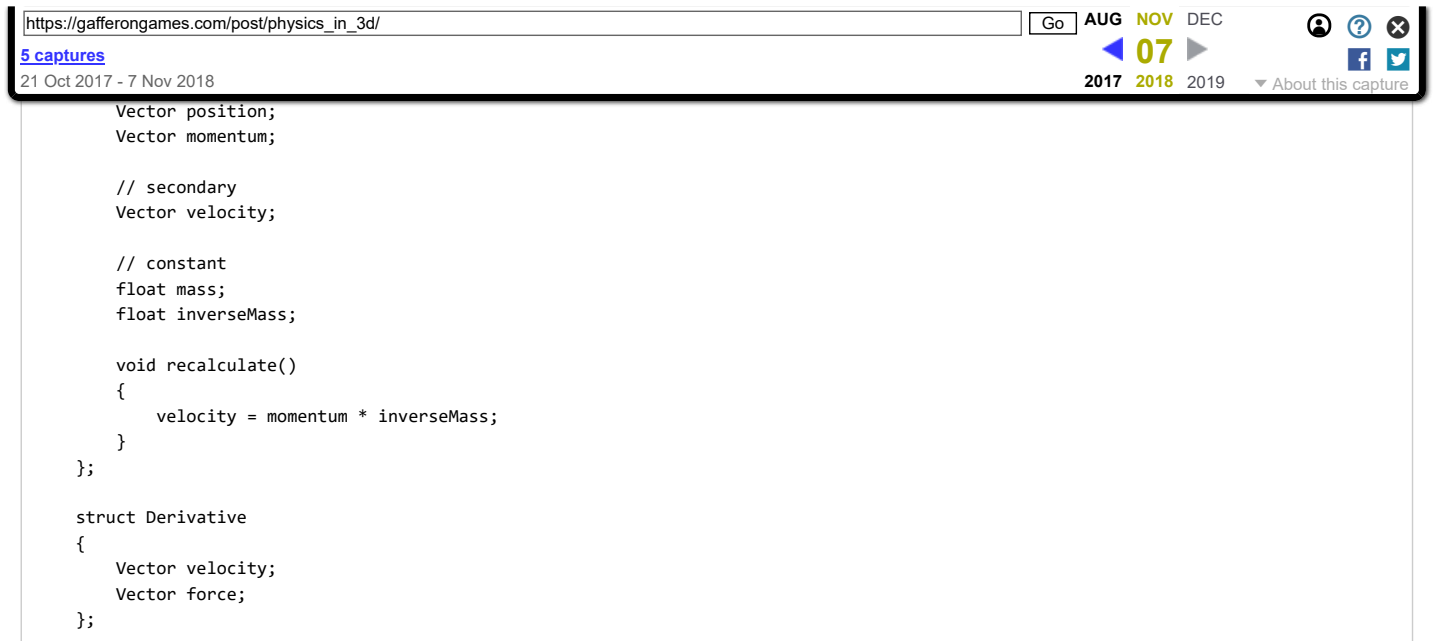
There are two ways we can do this. First, we can divide force by mass to get acceleration, then integrate this acceleration to get the velocity, and integrate velocity to get position.

The second way is to integrate force directly to get momentum, then convert this momentum to velocity by dividing it by mass, then finally integrate velocity to get position. Remember that momentum is just velocity multiplied by mass:

```
dp/dt = F
v = p/m
dx/dt = v
```

Both methods work, but the second way is more consistent with the way that we must approach rotation later in the article, so we'll use that.

When we switch to momentum we need to make sure that the velocity is recalculated after each integration by dividing momentum by mass. Doing this manually everywhere that momentum is changed would be error prone, so we now separate all our state quantities into primary, secondary and constant values, and add a method called 'recalculate' to the State struct which is responsible for updating all the secondary values from the primary ones:



If we make sure that recalculate is called whenever any of the primary values change, then our secondary values will always stay in sync. This may seem like overkill just to handle converting momentum to velocity, but as our simulation becomes more complex we will have many more secondary values, so it is important to design a system that handles this.

Spinning Around

So far we have covered linear motion, we can simulate an rigid body so that it moves in 3D space, but it cannot rotate yet.

The good news is that rotational equivalents to force, momentum, velocity, position and mass exist, and once we understand how they work, integration of rotational physics state can be performed using our RK4 integrator.

Let's start off by talking about how rigid bodies rotate. Because our objects are rigid they cannot deform. This means that we can treat the linear and rotational parts of an object's motion as being entirely separate: a linear component (position, velocity, momentum, mass) and a rotational component rotating about the center of mass.

How do we represent how the object is rotating? If you think about it a bit, you'll realize that for a rigid body rotation can only ever be around a single axis, so the first thing we need to know is what that axis is. We can represent this axis with a unit length vector. Next we need to know how fast the object is rotating about this axis in radians per second.

If we know the center of mass of the object, the axis of rotation, and the speed of rotation then we have all the information we need to describe how it is rotating.

The standard way of representing rotation over time is by combining the axis and the speed of rotation into a single vector called angular velocity. The length of the angular velocity vector is the speed of rotation in radians while the direction of the vector indicates the axis of rotation. For example, an angular velocity of $(2\pi, 0, 0)$ indicates a rotation about the x axis doing one revolution per second.

point your thumb down the axis, your fingers curl in the direction of rotation. If your 3D engine uses a left handed coordinate system then just use your left hand instead.

Why do we combine the axis and rate of rotation into a single vector? Doing so gives us a single vector quantity that is easy to manipulate just like velocity for linear motion. We can easily add and subtract changes to angular velocity to change how the object is rotating just like we can add and subtract from linear velocity. If we stuck with a unit length vector and scalar for rotation speed then it would be much more complicated to apply these changes.

But there is one very important difference between linear and angular velocity. Unlike linear velocity, there is no guarantee that angular velocity will remain constant over time in the absence of forces. In other words, angular momentum is conserved while angular velocity is not. This means that we cannot trust angular velocity as a primary value and we need to use angular momentum instead.

Angular Momentum, Inertia and Torque

Just as velocity and momentum are related by mass in linear motion, angular velocity and angular momentum are related by a quantity called the rotational inertia. This tensor is a measurement of how much effort it takes to spin an object around an axis. It depends on both the shape of the object and how much it weighs.

In the general case, rotational inertia is represented by a 3x3 matrix called an inertia tensor. Here we make a simplifying assumption by discussing physics in the context of simulating a cube. Because of the symmetries of the cube, we only need a single value for the rotational inertia: $\frac{1}{6} \times \text{size}^2 \times \text{mass}$, where size is the length of the sides of the cube.

Just as we integrate linear momentum from force, we integrate angular momentum directly from the rotational equivalent of force called torque. You can think of torque just like a force, except that when it is applied it induces a rotation around an axis in the direction of torque vector rather than accelerating the object linearly. For example, a torque of (1,0,0) would cause a stationary object to start rotating about the x axis.

Once we have angular momentum integrated, we multiply it by the inverse of the rotational inertia to get the angular velocity, and using this angular velocity we integrate to get the rotational equivalent of position called orientation.

However, as we will see, integrating orientation from angular velocity is a bit more complicated!

Orientation in 3D

This complexity is due to the difficulty of representing orientations in three dimensions.

In two dimensions orientations are easy, you just keep track of an angle in radians and you are done. In three dimensions it becomes much more complex. It turns out that you must either use 3x3 rotation matrices or quaternions to correctly represent the orientation of an object.

https://gafferongames.com/post/physics_in_3d/

Go

AUG NOV DEC

07

2017 2018 2019

About this capture

5 captures

21 Oct 2017 - 7 Nov 2018

orientation to get smooth framerate independent animation as per the time stepping scheme outlined in the [previous article](#)

(http://web.archive.org/web/20181107181511/https://gafferongames.com/post/fix_your_timestep/).

Now there are plenty of resources on the internet which explain what quaternions are and how unit length quaternions are used to represent rotations in three dimensions. Here is a particularly [nice one](#) (<http://web.archive.org/web/20181107181511/http://www.sjbrown.co.uk/quaternions.html>). What you need to know however is that, effectively, unit quaternions represent an axis of rotation and an amount of rotation about that axis. This may seem similar to our angular velocity, but quaternions are four dimensional vectors instead of three, so mathematically they are actually quite different!

We will represent quaternions in code as another struct:

```
struct Quaternion
{
    float w,x,y,z;
};
```

If we define the rotation of a quaternion as being relative to an initial orientation of the object (what we will later call body coordinates) then we can use this quaternion to represent the orientation of the object at any point in time. Now that we have decided on the representation to use for orientation, we need to integrate it over time so that the object rotates according to the angular velocity.

Integrating Orientation

We are now presented with a problem. Orientation is a quaternion but angular velocity is a vector. How can we integrate orientation from angular velocity when the two quantities are in different mathematical forms?

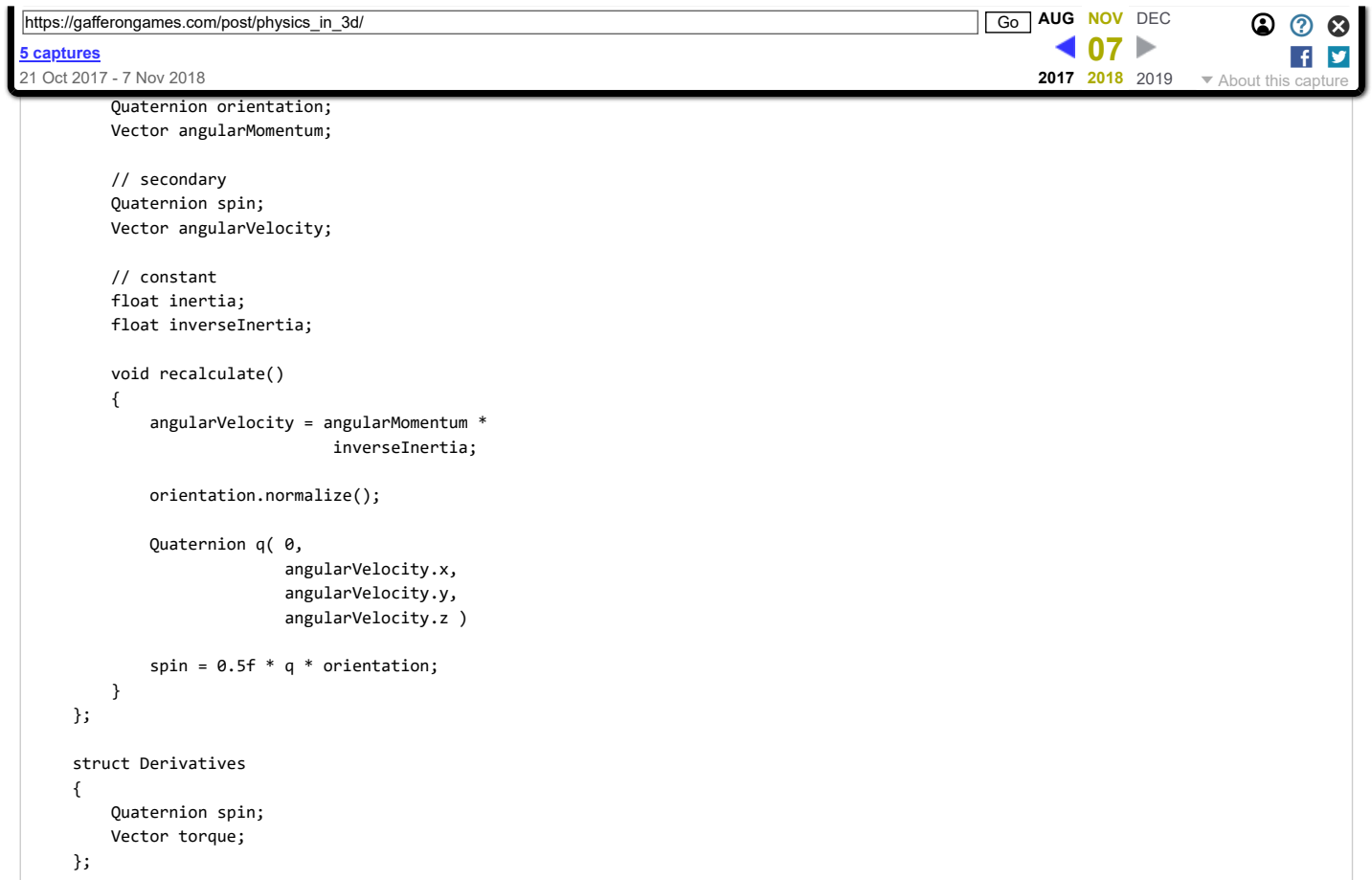
The solution is to convert angular velocity into a quaternion form, then to use this quaternion to integrate orientation. For lack of a better term I will call this time derivative of orientation “spin”. Exactly how to calculate this spin quaternion is described in detail [here](#) (<http://web.archive.org/web/20181107181511/https://www-2.cs.cmu.edu/~baraff/sigcourse/notesd1.pdf>).

Here is the final result:

$$d\mathbf{q}/dt = \text{spin} = 0.5 \cdot \mathbf{w} * \mathbf{q}$$

Where \mathbf{q} is the current orientation quaternion, and \mathbf{w} is the current angular velocity in quaternion form (0,x,y,z) such that x, y, z are the components of the angular velocity vector. Note that the multiplication done between \mathbf{w} and \mathbf{q} is quaternion multiplication.

To implement this in code we add spin as a new secondary quantity calculated from angular velocity in the recalculate method. We also add spin to the derivatives struct as it is the derivative of orientation:



Integrating a quaternion, just like integrating a vector, is as simple as doing the integration for each value separately. The only difference is that after integrating orientation we must renormalize the orientation quaternion to make it unit length, to ensure that it still represents a rotation.

This is required because errors in integration accumulate over time and make the quaternion ‘drift’ away from being unit length. I like to renormalize in the recalculate method for simplicity, but you can get away with doing it less frequently if cpu cycles are tight.

Now in order to drive the rotation of the object, we need a method that can calculate the torque applied given the current rotational state and time just like the force method we use when integrating linear motion. eg:

```

Vector torque( const State & state, double t )
{
    return Vector(1,0,0) - state.angularVelocity * 0.1f;
}

```

This function returns an acceleration torque to induce a spin around the x axis, but also applies a damping over time so that at a certain speed the accelerating and damping will cancel each other out. This is done so that the rotation will reach a certain rate and stay constant instead of getting faster and faster over time.

Combining Linear and Angular Motion

Now that we are able to integrate linear and rotational effects, how can they be combined into one simulation? The answer is to just integrate the linear and rotational physics state separately and everything works out. This is because the objects we are simulating are rigid so we can decompose their

https://gafferongames.com/post/physics_in_3d/

AUG NOV DEC
07
2017 2018 2019

5 captures
21 Oct 2017 - 7 Nov 2018

About this capture

Now that we have an object that is translating and rotating through three dimensional space, we need a way to keep track of where it is. We must now introduce the concepts of body coordinates and world coordinates.

Think of body coordinates in terms of the object in a convenient layout, for example its center of mass would be at the origin (0,0,0) and it would be oriented in the simplest way possible. In the case of the simulation that accompanies this article, in body space the cube is oriented so that it lines up with the x, y and z axes and the center of the cube is at the origin.

The important thing to understand is that the object remains stationary in body space, and is transformed into world space using a combination of translation and rotation operations which put it in the correct position and orientation for rendering. When you see the cube animating on screen it is because it is being drawn in world space using the body to world transformation.

We have the raw materials to implement this transform from body coordinates into world coordinates in the position vector and the orientation quaternion. The trick to combining the two is to convert each of them into 4x4 matrix form which is capable of representing both rotation and translation. Then we combine the two transformations into a single matrix by multiplication. This combined matrix has the effect of first rotating the cube around the origin to get the correct orientation, then translating the cube to the correct position in world space. See [this article](http://web.archive.org/web/20181107181511/https://www.gamedev.net/reference/articles/article695.asp) (<http://web.archive.org/web/20181107181511/https://www.gamedev.net/reference/articles/article695.asp>) for details on how this is done.

If we then invert this matrix we get one that has the opposite effect, it transforms points in world coordinates into the body coordinates of the object. Once we have both these matrices we have the ability to convert points from body to world coordinates and back again which is very handy. These two matrices become new secondary values calculated in the 'recalculate' method from the orientation quaternion and position vector.

Forces and Torques

We can apply separate forces and torques to an object individually, but we know from real life that if we push an object it usually makes it both move and rotate. So how can we break down a force applied at a point on the object into a linear force which causes a change in momentum, and a torque which changes angular momentum?

Given that our object is a rigid body, what actually happens here is that the entire force applied at the point is applied linearly, plus a torque is also generated based on the cross product of the force vector and the point on the object relative to the center of mass of the object:

$$\begin{aligned} F_{\text{linear}} &= \mathbf{F} \\ F_{\text{torque}} &= \mathbf{F} \times (\mathbf{p} - \mathbf{x}) \end{aligned}$$

Where \mathbf{F} is the force being applied at point \mathbf{p} in world coordinates, and \mathbf{x} is the center of mass of the object.

https://gafferongames.com/post/physics_in_3d/

AUG NOV DEC
07
2017 2018 2019

5 captures
21 Oct 2017 - 7 Nov 2018

About this capture

What is happening here is our everyday experience with objects clouding the true behavior of an object under ideal conditions.

Remember your pushbike when you were a kid? You would have to change your tire and flip the bike upside down. You could spin the tire around by pushing on it. You don't see any linear motion here, just rotation, so what is going on? The answer of course is that the axle of the wheel is counteracting the linear component of the force you applied, leaving only the rotational component. Not convinced? Imagine what would happen if you tried to ride your bike without an axle in your wheel...

Another example: consider a bowling ball lying on a slippery surface such as ice so that no significant friction is present. Now in your mind try to work out a way that you can apply a force at a single point on the surface of the bowling ball such that it will stay completely still while rotating on the spot. There is no way you can do this! Any point where you push would also make the bowling ball move linearly as well as rotate. To apply a pure rotation you'd have to push on both sides of the ball, canceling the linear component of your force out leaving only torque.

So remember, whenever you apply a force to an object there will always be a linear force component which causes the object to accelerate linearly, as well as, depending on the direction of the force, a rotational component that causes the object to rotate.

Velocity at a Point

The final piece of the puzzle is how to calculate the velocity of a single point in the rigid body. To do this we start with the linear velocity of the object, because all points must move with this velocity to keep it rigid, then add the velocity at the point due to rotation.

This velocity due to rotation will not be constant for every point in the body if it is rotating, as each point in the body must be spinning around the axis of rotation. Combining the linear and angular velocities, the total velocity of a point in the rigid body is:

$$\mathbf{v}_{\text{point}} = \mathbf{v}_{\text{linear}} + \mathbf{v}_{\text{angular}} \text{ CROSS } (\mathbf{p} - \mathbf{x})$$

Where \mathbf{p} is the point on the rigid body and \mathbf{x} is the center of mass of the object.

Conclusion

We have covered the techniques required to simulate linear and rotational movement of a rigid body in three dimensions. By combining the linear and rotational physics into a single physics state and integrating, we can simulate the motion of a rigid body in three dimensions.

NEXT ARTICLE: [Spring Physics](#)

(http://web.archive.org/web/20181107181511/https://gafferongames.com/post/spring_physics/)

https://gafferongames.com/post/physics_in_3d/

Go

AUG

NOV

DEC

07

2017

2018

2019

About this capture

5 captures



21 Oct 2017 - 7 Nov 2018

?

×

f

t

-  (<http://web.archive.org/web/20181107181511/https://twitter.com/gafferongames>)
-  (<http://web.archive.org/web/20181107181511/https://github.com/gafferongames>)

Copyright © Glenn Fiedler, 2004 - 2018