Physics - Angular motion

Introduction

Our physics engine can now move objects around in a physically accurate way - at least in straight lines, anyway. As well as linear motion, rigid bodies can also spin around, changing their orientation over time. In this tutorial, we'll see how to apply the correct forces to allow this rotational movement to happen, and adjust the object pushing code we introduced in the previous tutorial to allow us to make object's twist and spin as they are clicked with the mouse.

New Terms

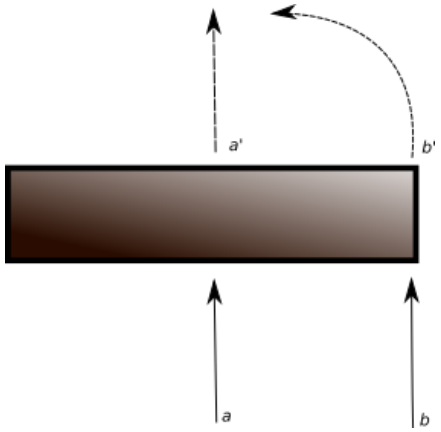
Torque

We saw in the previous tutorial that the forces imparted upon an object resulting in a change of *velocity*, with the magnitude of the change (the *acceleration*) being related to the object's (inverse) mass:

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

From this equation, we can see how to change an object's *linear velocity*; how it moves in a straight line. As the physics simulations we wish to create get more complex, we probably also want objects to rotate, and spin around when colliding with other objects. To do this, we need to determine how much *angular velocity* the physics object has, and integrate it accordingly from the angular acceleration. Just as linear acceleration is applied via *force*, angular acceleration is applied via *torque*.

We can, if we want, add torque to objects directly, by adding an angular force every frame - the wheels on a car might be modeled this way. Things get more interesting if we consider the other cases in which twisting motion could be applied to an object. Consider the following two cases, of a cuboid object being 'pushed' by a force applied at points *a* or *b*:



How would the object move under each of the shown forces? Assuming that we're dealing with a rigid body, with an even distribution of mass throughout its volume, it's fairly intuitive that if the object has a force applied at point *a*, its resulting movement would be similar to *a'* - it would move in a straight line. At point *b* though, the object should twist around, with the corner of the cuboid following a path similar to *b'*. You can test this out by flicking around a pen on your desk, you'll see that the further away from the middle of the pen you flick, the more the pen wants to rotate as it

moves, so our example cuboid should really start rotating - the force applied has added *torque* to the object, resulting in *angular* velocity, as well as linear velocity.

Determining the amount of torque that a force applies to an object is fairly straightforward. If we are applying a force $\mathbf{F}$ at a relative position of d from the centre of mass of the object, the amount of torque τ is defined simply as:

$$\tau = d \times F$$

Where $\times$ is the *cross product* between the two vectors. As you saw in the earlier graphics module, the cross product between two vectors produces a vector that is orthogonal to both - in this example as d is directed along the x axis, and $\mathbf{F}$ along the z axis, it will produce a vector directed along the y axis. This results in an amount of rotation *around* this axis, causing our example object to spin around, changing its yaw, by an amount proportional to the magnitude of the torque vector, and the object's mass.

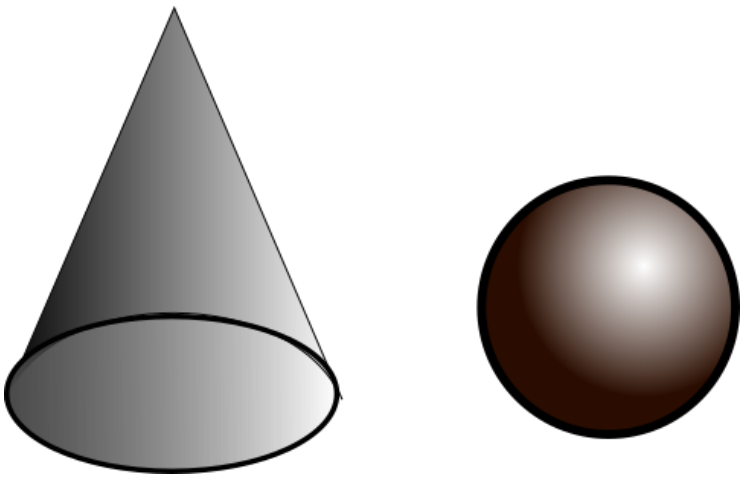
Just as the unit of force is a Newton, the unit of torque is the *Newton Metre* - the result of applying a force at 1 metre from the object's centre of mass.

Inertia

The acceleration of an object is related to the force acted upon it, and the object's mass. Given that, shouldn't the torque of an object be influenced by its mass, too? Indeed it should! However its not quite as easy as with linear motion. When we push an object at its centre of mass (that is, we're applying a purely *linear* force), we're trying to move the whole object at once at the same rate. When we push an object with some rotational force (like the cuboid example earlier), the amount of force required to rotate the object on a particular axis depends on the distribution of mass across its volume - we're trying to make the mass around the outsides of that cuboid move more than the mass in its middle. Therefore, to determine the amount of angular motion a force should impart on an object, we need more than just a scalar mass value, but something that can describe the distribution of that mass around the object's volume in each axis - the *moment of inertia*. This quantity describes how resistant an object is to changes in its angular velocity, much as mass limits changes in linear velocity.

Inertia Tensor

We can represent an object's moment of inertia using a special type of matrix, known as a *tensor*. To understand why we can't just have a single scalar value for our moment of inertia, consider the following shapes:



If we assume that both are solid shapes, made of a single, uniform material, then it should be clear that the cone has a different distribution of mass (mostly around the bottom of the shape) compared the sphere (evenly distributed).

Common Moments of Inertia

For simple symmetrical objects, we usually don't have to do anything *too* laborious to determine the inertia tensor; There's a number of standard calculations for objects shaped like the standard collision volumes we would use in a game physics simulation.

Solid Sphere

$$I = \frac{2}{5}mr^2$$
$$\begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{bmatrix}$$

Solid Cuboid

The inertia tensor for a cuboid is a little more complex, as the distribution of mass depends upon the width, height, and length of the cuboid. We can represent this using the following inertia tensor:

$$\begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix}$$

Where I_x, I_x , and I_x are the following:

$$I_x = \frac{1}{12}m(z^2 + y^2)$$
$$I_y = \frac{1}{12}m(x^2 + z^2)$$
$$I_z = \frac{1}{12}m(x^2 + y^2)$$

As before, m is the mass of the object, and x,y and z are the width, height, and length of the cuboid.

Solid Cone

While a fairly unusual shape, it might be useful to take a closer look at the cone example from earlier - exactly how does the angular force change depending on the shape of the object? First off, let's take a look at the moment of inertia for a cone, which assume's the object's central point as at the point of the cone:

$$\begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix}$$

Where I_x, I_x , and I_x are the following:

$$I_x = I_z = \frac{3}{20}mr^2 + \frac{1}{10}mh^2$$
$$I_y = \frac{3}{10}mr^2$$

So, if our cone's mass is 1, it's base has a radius of 1, and its peak is 1 unit high, we'd get the following tensor:

$$\begin{bmatrix} 0.25 & 0 & 0 \\ 0 & 0.3 & 0 \\ 0 & 0 & 0.25 \end{bmatrix}$$

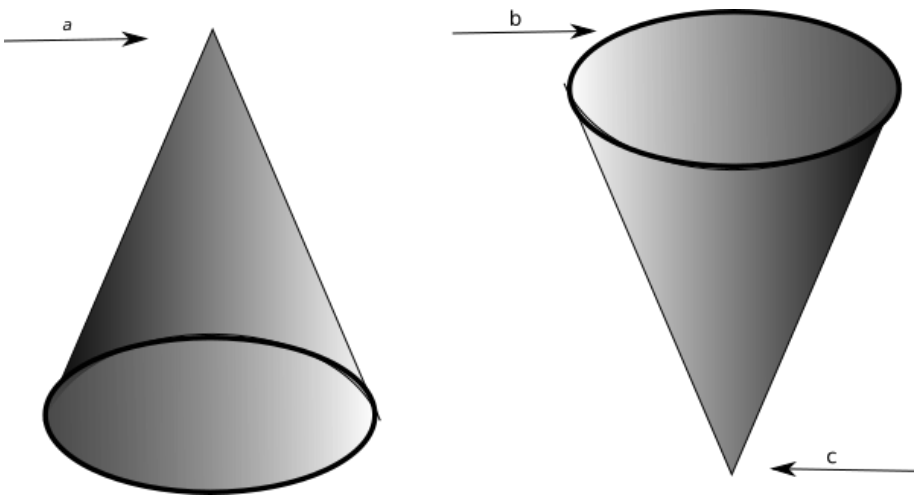
While we can visually see that a cone has more mass distributed along x and z, we can now see how this results in a scaling matrix with different amounts of scaling on the axes where most of the mass is situated in the shape.

Inverse inertia tensor

In the previous tutorial, we saw that rather than just mass, we could save computation (and get the ability to naturally represent 'infinitely massful' objects that we don't want to move) by using the *inverse* mass of an object in our calculations. For the same reasons, we can do the same with the moment of inertia, and instead create an *inverse* inertia tensor I^{-1}. For moments of inertia like the ones above, this is as simple as using the *reciprocal* of each of the values down the diagonal of the tensor.

Rotating the Inertia Tensor

You might be looking at the moments of inertia encoded as tensors above, and still be wondering exactly *why* we're using a matrix to represent them, rather than a vector, if all of the values end up down the matrix diagonal. There's two reasons - for one, our object may not be symmetrical, and therefore requiring a much more complex calculation using integrals (imagine slicing up the asymmetric object up into little cubes, and seeing what proportion of those cubes are along each 3D axis, and then in each axis, how those proportions change when moving to each other axis), that results in a 3x3 matrix that describes the distribution between each axis. Secondly - we apply forces in *world space* (that is, a force of (10,0,0) should push an object along the global x axis), but the moment of inertia describes the distribution in a space local to that of the object. For objects without a uniform distribution of mass (like our cone example earlier), this becomes important, as their rotational response to forces should be consistent - the same relative force applied to that cuboid should have the same effect, no matter what orientation the cuboid is in:



Forces a and c should produce the same effect on the cone (assuming they are the same magnitude), whereas force b, even if it were the same magnitude as the others, should have less of an effect over the cone, as the distribution of mass throughout its volume is different.

One way to solve this would be to, for every force applied to an object in a frame, transform that force by the inverse of the object's orientation (to bring it into local space), then multiply it by the tensor to determine the effect of inertia, before multiplying the result back into world space by multiplying by the orientation again. That works...but what if there's a lot of forces added to the object in a frame? That's a lot of space transformations to apply! Instead, we can rotate the inertia tensor of the object by the orientation of the object once per frame, bringing it from the local space of the object into the world space of our simulation, and therefore suitable for however many forces will be applied to the object in that frame.

Angular Velocity

Now that we understand why we have a matrix for the inertia tensor, we can see how to use it to actually determine how much angular velocity we gain from torque. It's the exact same process as with linear velocity - except this time we multiply torque by the inverse interia tensor (rather than

inverse mass by acceleration). From our torque force amount τ, we can determine the amount of angular acceleration α, and then integrate that into angular velocity ω:

$$\alpha = I^{-1}\tau$$
$$\omega = \alpha dt$$

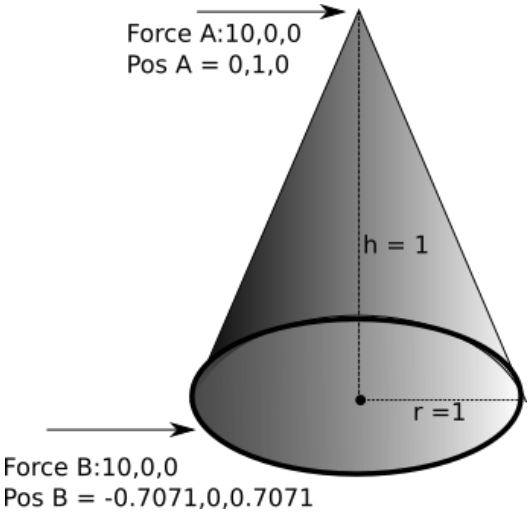
The number of symbols is increasing! In code, though, all of this is fairly easy to implement, and in the tutorial code we can keep using more descriptive names for each of the concepts as they are expanded upon into our complete physics engine.

Cone example

To get a clearer picture of the use of the inertia tensor, and how it changes the angular acceleration applied due to torque, let's revisit the cone inertia tensor outlined earlier, and its inverse:

$$\mathrm{I} = \begin{bmatrix} 0.25 & 0 & 0 \\ 0 & 0.3 & 0 \\ 0 & 0 & 0.25 \end{bmatrix} I^{-1} = \begin{bmatrix} 4 & 0 & 0 \\ 0 & 3.333 & 0 \\ 0 & 0 & 4 \end{bmatrix}$$

Now let's see how that tensor would change the amount of torque applied at two different points, shown in this diagram:



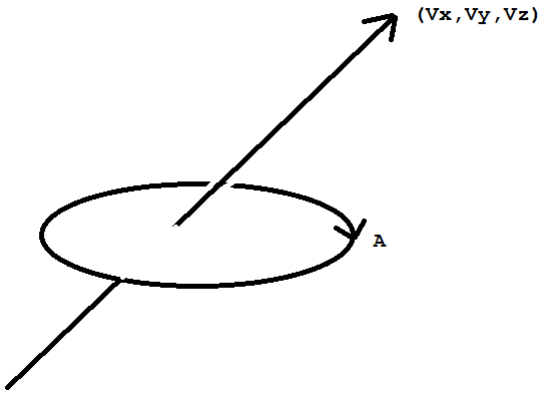
Each force is being applied at a point is 1 unit away from the object's position, just along a different axis. Using these vectors, we can see if we applied a force at point A, we'd get a resulting torque of (0,0,10) (the result of the cross product of force a and position a). At point B we'd get a torque of (0,0,-7.071), giving us two very different axes of rotation, and different *amounts* of rotation around those axes. So far, these calculations haven't used our inertia tensor, and so can't really describe how much rotation those torques will actually apply - mass will resist attempts to change the momentum of the object, so the distribution of the mass is important. After transforming each of these vectors by the inertia tensor above, we get the resulting angular acceleration α:

$$\text{Result of force A} = (0, 0, 40) = I^{-1}(0, 0, 10)$$
$$\text{Result of force B} = (0, -23.6, 0) = I^{-1}(0, -7.071, 0)$$

While both applied forces were of the same magnitude, they were applied at different positions on the volume, and thus resulted in different amounts of torque, which when scaled by the inverse inertia tensor from earlier, lets us see that the force applied at point A causes much more rotation than the force applied at point B - there's less mass at that point to resist the change.

Quaternions

In the graphics module, if you examined the code for skeletal animation, you will have come across *quaternions*. A quaternion is an efficient way of storing an orientation, that also has the handy property of a well formed method by which to rotate an orientation. Unlike a rotation matrix, which would have 9 elements (or 16 if it were made homogeneous) a quaternion can be store an orientation using just 4 values. It does this by encoding within it an axis, and the rotation around it:



To integrate the angular velocity into an object's orientation, we need to form a quaternion that represents the amount we wish to rotate by, and then use quaternion multiplication to form a new quaternion that is the combination of both the old orientation and the amount to move by. You might therefore think that the 4 values would be a normalised direction vector to represent the axis, and a scalar to represent the rotation around it, but its a bit trickier than that. We want a way to both represent an orientation, and also a *change* in that orientation, so we need a way of combining and changing the axis and angle in a consistent manner. Rather than the axis and angle directly, the 4 values of a quaternion are formed from a vector V and angle A as follows:

$$\begin{aligned} Q^x &= sin(A/2) * Vx \\ Q^y &= sin(A/2) * Vy \\ Q^z &= sin(A/2) * Vz \\ Q^w &= cos(A/2) \end{aligned}$$

This form has a few useful properties. For, one we can invert the orientation of quaternion Q taking its *conjugate*:

$$Q^{-1} = [-Q^x, -Q^y, -Q^z, Q^w]$$

Just like a transformation matrix, the inverse of a quaternion moves us back the other way - if we rotate a vector by quaternion Q, and then by quaternion Q^{-1}, we'd end up back where we started.

To construct a quaternion Q_3 that encodes both a transformation of Q_1 and Q_2, we must multiply those quaternions, just the same as with matrices. Quaternion multiplication looks like this:

$$\begin{aligned} Q_3^x &= (Q_1^x \cdot Q_2^w) + (Q_1^w \cdot Q_2^x) + (Q_1^y \cdot Q_2^z) - (Q_1^z \cdot Q_2^y) \\ Q_3^y &= (Q_1^y \cdot Q_2^w) + (Q_1^w \cdot Q_2^y) + (Q_1^z \cdot Q_2^x) - (Q_1^x \cdot Q_2^z) \\ Q_3^z &= (Q_1^z \cdot Q_2^w) + (Q_1^w \cdot Q_2^z) + (Q_1^x \cdot Q_2^y) - (Q_1^y \cdot Q_2^x) \\ Q_3^w &= (Q_1^w \cdot Q_2^w) - (Q_1^x \cdot Q_2^x) - (Q_1^y \cdot Q_2^y) - (Q_1^z \cdot Q_2^z) \end{aligned}$$

As you can possibly tell from the above formula, quaternion multiplication is not *commutative*, that is Q_1 x Q_2 doesn't give us the same answer as Q_2 x Q_1 - this should be familiar to you, as transformation matrices work the same way.

On the topic of transformation matrices, it's possible to expand out a quaternion into a 3x3 rotation matrix, and so also a full 4x4 homogeneous matrix, so we don't 'lose' anything by storing orientations as a quaternion, but we do gain an easy way of keep it separate from the scaling of an object.

Integrating Angular Velocity

The final thing we need to do with a quaternion, is see how to transform an axis angle representation into a quaternion. Why? Think about our angular velocity from earlier - we store it as a vector, which represents how much the object is twisting around that axis. We need to do two things: first, we need to work out how much this angular velocity adds per timestep, and secondly we need to turn the angular axis vector into a quaternion. We can do both things at once, with the following operations:

$$V_{temp} = \frac{dt(velocity_{ang})}{2}$$
$$Q_{temp} = [V_{temp}x, V_{temp}y, V_{temp}z, 0] \cdot orientation$$
$$orientation = \|orientation + Q_{temp}\|$$

V_{temp} holds our relative change in orientation for the current timestep, and Q_{temp} transforms this into what's known as a *pure* quaternion - one with no fourth element, relative to our original orientation. It looks very confusing, and relies on the multiplicative property of quaternions, plus a normalisation, to work. Quaternions are one of those things that as game engineers we like due to their efficiency, but also dislike due to the obtuse nature of their operation - we won't need to do any more with them in this module, so all we need to remember is that they're efficient representations of orientation.

Tutorial Code

To demonstrate angular motion on our rigid bodies, we're going to modify the previous tutorial, so that the forces applied on the mouse click impart the correct amount of torque for the position at which we click; this should allow the shapes to spin about on screen.

Integrating torque

To adjust the orientation of our object's over time, we need to integrate the applied torque into the object's angular velocity. To do this, we are going to modify the *IntegrateAccel* function, by adding the following code after the *SetLinearVelocity* call we added in the previous tutorial:

```
1  //We start adding new code after this existing line:
2        object->SetLinearVelocity(linearVel); //previous code
3
4        //Angular stuff
5        Vector3 torque = object->GetTorque();
6        Vector3 angVel = object->GetAngularVelocity();
7
8        object->UpdateInertiaTensor(); //update tensor vs orientation
9
10       Vector3 angAccel = object->GetInertiaTensor() * torque;
11
12       angVel += angAccel * dt; //integrate angular accel!
13       object->SetAngularVelocity(angVel);
14    }
15 }
```

PhysicsSystem::IntegrateAccel method

Much as with linear velocity, we calculate an acceleration value from the object's inverse mass - except this time, we're using the object's *inertia tensor* to transform the acceleration (line 10). As part of this process, we'll also update the object's inertia tensor, based on its current orientation - remember, we need to rotate the inertia tensor so that its values accurately reflect the object's current orientation. Once this is done, we can just update the angular velocity with the current angular acceleration, scaled by the timestep *dt* (line 12).

Integrating angular velocity

With angular acceleration integrated, we can move on to changing the *IntegrateVelocity* method to correctly update the object's orientation according to its angular velocity. In the *IntegrateVelocity* method, add the following code after the call to *SetLinearVelocity*:

```
1  //We start adding new code after this existing line:
2        object->SetLinearVelocity(linearVel);
3
4        //Orientation Stuff
5        Quaternion orientation = transform.GetLocalOrientation();
6        Vector3 angVel = object->GetAngularVelocity();
7
8        orientation = orientation +
9           (Quaternion(angVel * dt * 0.5f, 0.0f) * orientation);
10       orientation.Normalise();
11
12       transform.SetLocalOrientation(orientation);
13
14       //Damp the angular velocity too
15       angVel = angVel * frameDamping;
16       object->SetAngularVelocity(angVel);
17    }
18 }
```

PhysicsSystem::IntegrateVelocity method

The actual integration occurs on line 9. The 0.5 is due to how quaternions work - trust that we aren't 'losing' any angular velocity. Most of this method is centered around getting data in and out of that orientation quaternion As with linear velocity, we're also going to apply some damping (line 15) so that objects don't just spin forever.

Applying forces

To add torque to a physics object, we're going to need a new method in the **PhysicsObject** class, *AddForceAtPosition*:

```
1  void PhysicsObject::AddForceAtPosition(
2           const Vector3& addedForce, const Vector3& position) {
3     Vector3 localPos = position - transform->GetWorldPosition();
4
5     force  += addedForce;
6     torque += Vector3::Cross(localPos, addedForce);
7  }
```

PhysicsObject::AddForceAtPosition method

This demonstrates in code the calculation of torque - from the passed in world position variable, we need to calculate the position relative to the object's centre of mass (line 3), and then use the cross product to determine the axis around which this force will cause the object to spin. Remember that the cross product produces flipped values depending on the order of parameters, so be careful when using it.

MoveSelectedObject method

To use the new *AddForceAtPosition* method, we need to modify the *MoveSelectedObject* of the *TutorialGame* class. Instead of calling AddForce, we're instead going to call *AddForceAtPosition*, and pass it the collision point in the world as the second parameter:

```
if (closestCollision.node == selectionObject) {
   selectionObject->GetPhysicsObject()->AddForceAtPosition(
         ray.GetDirection() * forceMagnitude,
         closestCollision.collidedAt);
}
```

TutorialGame::MoveSelectedObject method

Conclusion

If we click on objects now, they should get pushed around such that they slightly twist around under the application of torque. This is dependent on where the objects are clicked - remember, the further away from the centre of mass we click, the more the force we impart upon the object is applied as a twisting motion.

Our objects are starting to act a little more like how 'real' objects would, but they aren't perfect! We can currently push objects into the floor, and into each other, with no consequence. In the next part of the tutorial series, we'll take a look at how to remedy this using methods for *collision detection*, and *collision resolution*, which together allow us to determine where an object has touched another one, and what this will do to the linear and angular motion of the colliding objects.

Further Work

1) To accentuate the affect of torque on our shapes, and to demonstrate how the inertia tensor affects our objects, try making cube shapes that are not uniform on each axis, and try applying forces to them at varying points with mouse clicks - you should be able to make objects feel 'heavier' with a change in mass, and a force applied around some axes should have a noticeably differenty result than others.

2) Hollow spheres have a different inertia tensor than solid spheres (their mass is distributed away from the origin). Investigate what this inertia tensor might be, and try modifying the TutorialGame::AddSphereToWorld method to allow choosing whether the sphere is hollow or solid.