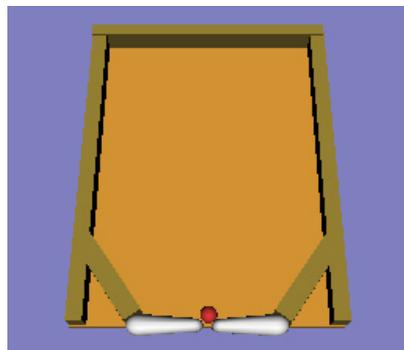




Pinball Tutorial

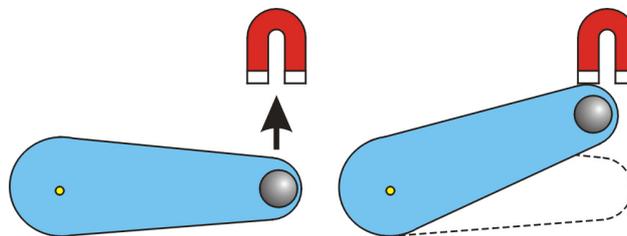
The pinball tutorial is a good example of user control over a physical object. We want to create a simulation of a pinball table with walls, 2 flippers and a ball. In a typical 3D application you would expect to apply a transformation to each flipper when the user activates the flipper and then check the position of the ball, applying a force and attempting to work out spin etc. to give some form of realistic result.



With physics engines the approach is somewhat different. Rather than attempt direct control over the object, approach the problem from more of an engineering view: in order to flip the flipper we need (ideally) to apply forces, impulses or torques to cause the flipper to flip (i.e. don't think about directly moving the flipper, but introducing a force to cause the flipper to move). The general rule is:

Always try to get the physics engine to cause the motion you're looking for

So in this case, a flipper is normally driven by some form of drive shaft attached to a stepper motor or a magnetic movement attached to the flipper drive shaft. We'll use the latter approach: what we're going to do is introduce a "magnetic" pull to pull the flipper into place when activated.

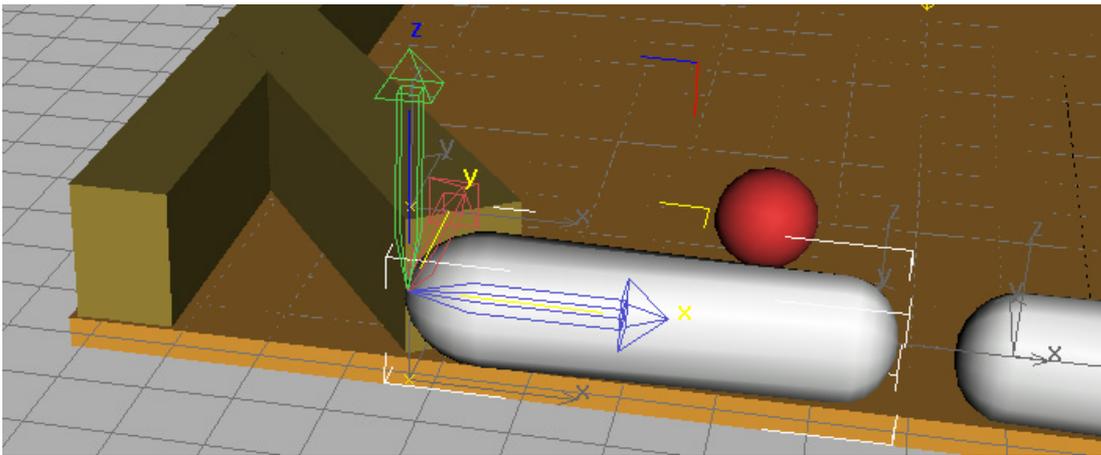


To achieve this we need 3 things:

1. The magnet must be fixed in some way so that it pivots about the correct point.
2. When not activated the paddle should stay in the closed position.
3. When activated the paddle should switch quickly to the open position.

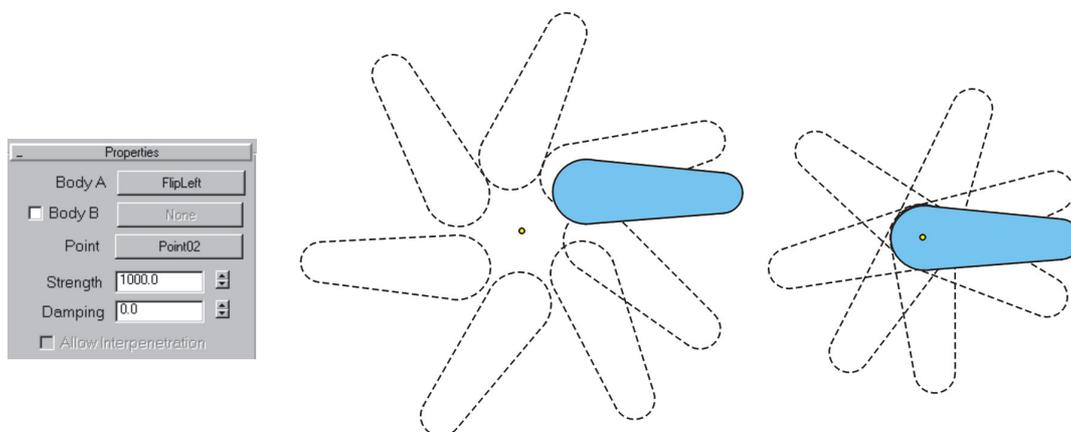
Creating the Flipper

To achieve all this, we construct the table, ball and flipper paddles as normal in max. To fix the pivot point of the flipper and to allow the flipper to rotate about a fixed axis (the z-axis of the flipper as shown in the picture below) we need to use a constraint, and **linear dashpots** are the best way to do this. Dashpots constrain points to stay together but what we're trying to do is create a revolute joint. This can be created using 2 dashpots.



If we constrained the flipper to this pivot point using a single dashpot, the flipper would be free to rotate about its x-axis (not what we want). By using 2 dashpots we create an approximation to a hinge. First a brief explanation of dashpots; consider dashpots to be like very stiff springs i.e. they attempt to take 2 points in space and make them occupy the same point in space. If these points are associated with objects, then those objects will be *constrained* in some way i.e. they can move about but with the restriction that the point on the object occupies the point in space that the dashpot connects it to.

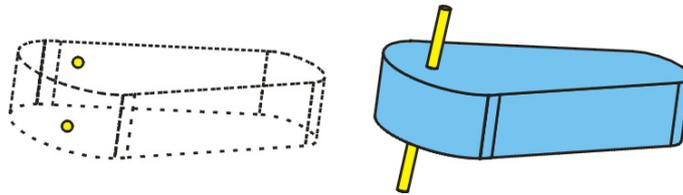
Time for some diagrams; in the picture below we show (in 2D) an object connected via a dashpot to a point in space (in max you do this by creating a point helper and specifying this in the dashpot parameters).



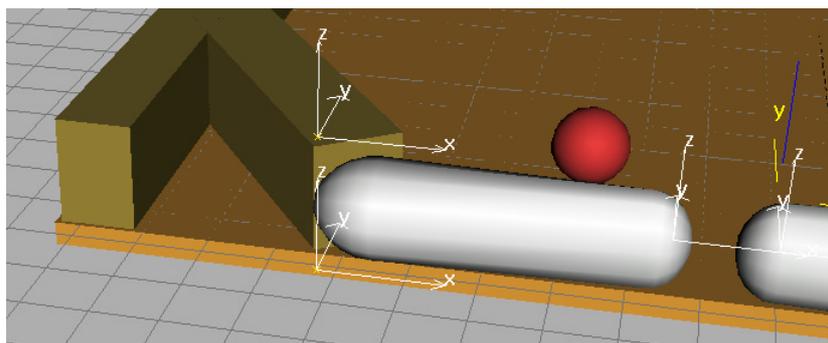
When attached to a point in space by a dashpot the object is free to move, as long as this point remains fixed – and it doesn't matter whether the point is on or off the object (officially the dashpot constrains the point specified in the local coordinate system of the object to the point in the world co-ordinate space). So if the point is on the object the object is free to spin around that point; if off the object then the same happens but it appears as if the object is connected to the point via a hidden line (think of it like you created a fixed line connecting the center of the object to the point wherever it is and that this line is rigidly connected to the object). The **strength** parameter refers to how rigidly the physics attempts to enforce this constraint, and the **damping** parameter may be used to remove wiggly behavior.

It is usually a good idea to keep the damping value less than 10% of the strength value. Values higher than this lead to unstable results. It's also a good idea to keep the strength parameter as low as possible – higher values lead to less stable solutions.

This is clearly not enough to create a hinge. With a single point dashpot, the flipper is free to rotate in any direction around the point, but we want it to rotate only about its z-axis. The solution is to constrain it using 2 dashpots to 2 different points located along the z-axis.



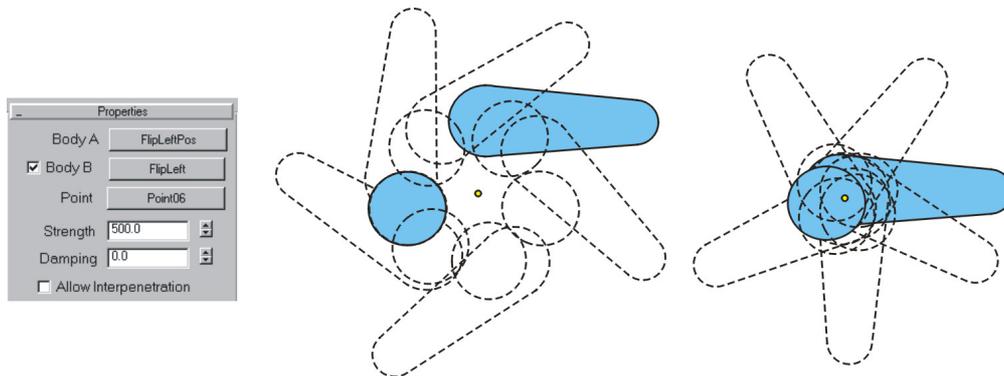
The 2 dashpots will ensure that the points on the flipper remain at the points in world space – this gives the result we're looking for. The flipper will line up along the line connecting the 2 points – these points can be placed at many different points along the flipper's z-axis (as long as they are equidistance from where you want the flipper to lie). Dashpots will be pretty strong, but if you push the end of the flipper it will cause the flipper to go off-axis and then spring back into position. In this case, though, we can use the table to stabilize the flipper (i.e. pushing down on the flipper will have no result because it will just hit the table).



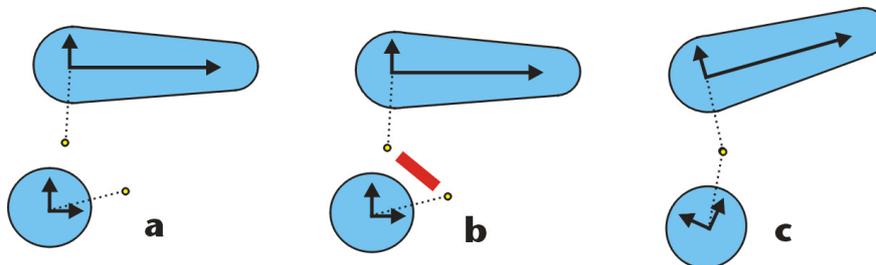
So for our flipper in max, create 2 **point helpers** in max long an axis that we want the flipper to spin around (as shown above) and fix the flipper object to these points via 2 linear dashpots.

Moving the Flipper

This is enough to create free moving flippers (i.e. they will spin around on their axes quite happily). The next stage is to constrain the movement of the flippers themselves (i.e. opening and closing in response to mouse button events). We're going to use dashpots again, but this time to connect objects to each other rather than to fixed points in space.



The picture above shows what happens when you involve a second object in a dashpot constraint. The object is no longer constrained to a point in world space but to a point on another object. You still need to specify a point to specify the connection between the 2 objects. The dashpot will always attempt to bring this point in each of the object's local coordinate system together – each of the objects is free to spin around this point. Again think of the objects being connected to the point by a fixed line (or better still, consider the point to be part of each object).

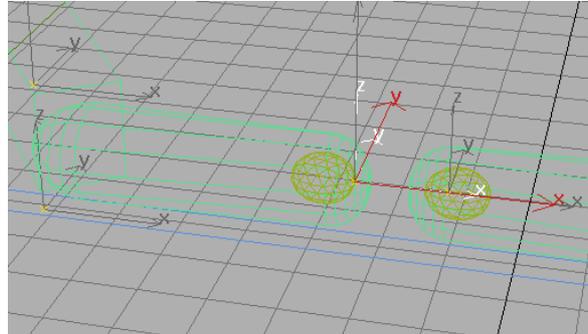


This is shown in the picture above. Imagine the dashpot was switched off – then the objects would be free to move normally. Shown here in **a** is where the point will move to in each of the object's local coordinate system (even though originally there was only a single point specified). In **b** we turn the dashpot back on which creates a force to try and bring the 2 points back into line. In **c** the dashpot has done its work and the 2 points are now back into position, but the objects have rotated slightly i.e. the objects are free to move so long as the 2 points stay together. These objects are not fixed in space but *only relative to each other*.

If the point is on each of the objects (as shown in the previous picture above on the right) then you could potentially have a problem as the objects will be interpenetrating, so usually you will select the **Allow Interpenetration** check box to disable collisions between this pair of objects).

Now we can address the control of the flippers. Rather than apply a force to push the flipper we want to specify 2 positions that it can take (and for the flipper to rapidly rotate to assume these positions, and in doing so bat away the ball if it's close). We do this by

constructing a dummy object (in this case a sphere called **FlipLeftPos**), which is not displayed, but used to pull the end of the flipper into place. We will attach the sphere to the end of the flipper using a dashpot.



Shown above is the construction in max. We have the dummy sphere placed inside the end of the flipper and a point helper located at the point where we want to connect the two objects. This point is at the center of the end of the flipper and on the edge of the sphere. The sphere will be able to pivot around this point creating a sort of joint between the sphere and flipper giving good freedom of movement. Now, if the sphere is moved, the flipper will be pulled into position.

The dummy sphere needs to be disabled in the display otherwise it will be visible. In fact in the current demo, we have not disabled the sphere (for simplicity). Use the **visibility** property of the associated models (i.e. set them to **#none**).

Pinball Action

The last piece of the puzzle is to move the dummy sphere in response to mouse input. When the flipper is not active (i.e. user has not pressed the mouse button) we fix the ball at one position pulling the flipper to its off position. When the user presses the mouse button we'll move the sphere up to pull the flipper into the open position.

We now need to fix the sphere at these positions. We do this by **pinning** it after moving it (and **unpinning** it just before moving it). If we didn't do this, the sphere would be free to move around under gravity and after being hit by other objects. We should also disable collisions with this sphere, but we'll leave that for another tutorial.

To begin with we determine the current position of the sphere (as saved by the modeler):

```
on beginSprite me
  pLeft = sprite(2).member.model("FlipLeftPos").transform.position
  pFirst = true
end
```

The first time entering the frame the 2 dummy spheres are pinned:

```
on enterFrame me
  if pFirst then
    sprite(2).pHavok.rigidBody("FlipLeftPos").pinned = true
    sprite(2).pHavok.rigidBody("FlipRightPos").pinned = true
    pFirst = false
  end if
end
```



When the user presses the mouse button (you may want to use keys on the Mac because of the lack of a second mouse button for the second paddle) we unpin the sphere, move it up a bit and then pin it again:

```
on mouseDown me
  sprite(2).pHavok.rigidBody("FlipLeftPos").pinned = false
  sprite(2).pHavok.rigidBody("FlipLeftPos").position = pLeft + \
    vector( 0, 25, 0 )
  sprite(2).pHavok.rigidBody("FlipLeftPos").pinned = true
end
```

Do something similar for the right flipper and you have a pinball flipper action. Next step is to add bumpers and traps etc. One trick is to set the restitution of bumper objects to be greater than 1 (i.e. the ball will bounce off faster than when it hit) and to use collision callbacks to create a scoring system. Have fun.