TI-nspire™

Scripting Tutorial - Lesson 24: (3.2) Welcome to the Physics Engine Part 2

Download supporting files for this tutorial

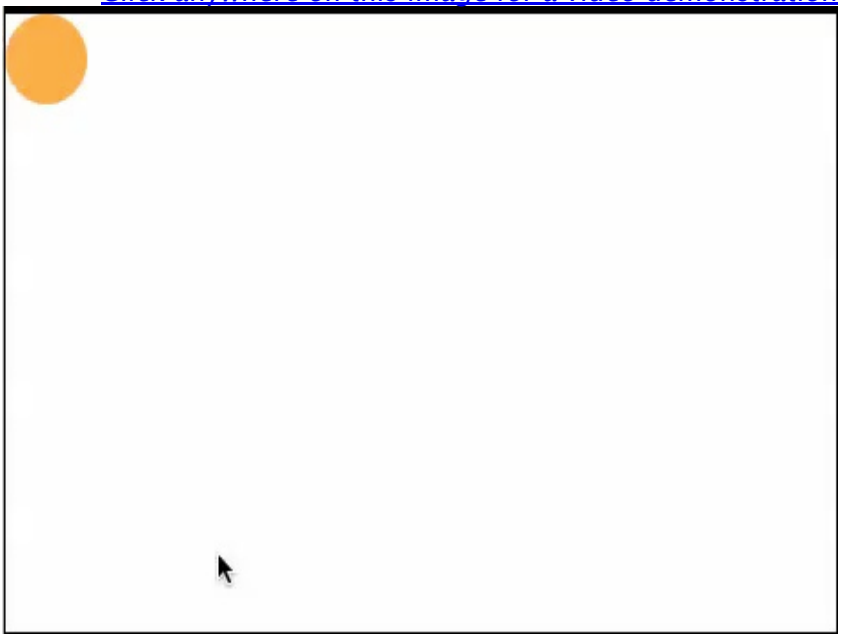
Download this page in PDF format

Texas Instruments TI-Nspire Scripting Support Page

Welcome to the Physics Engine Part 2 (3.2)

The previous lesson gave a useful Quick Start to the Physics Engine, but if we are going to use it to advantage, then not only are there many more commands to learn, but there are structural approaches to designing our scripts that we could and should take advantage of. This lesson approaches the same problem as previously - getting a ball bouncing around the screen - but in a more sustainable and effective way.

Click anywhere on this image for a video demonstration



1. The Big Picture: Looking at the Structure of our Script

Top of Page

If you have not already done so, copy the script from your browser window and paste it into the TI-Nspire Script Editor, Set it by pressing ctrl-s and enjoy the show. You should see an orange circle bouncing around the screen.

First observe that by defining all the variables up front as local then they will stay that way - and local variables are always much more efficient (performance-wise) than globals. They are also defined for all subsequent functions. Our previous approach - defining, say, w and h as local within the on.resize function means that they are only defined within that function and must be defined again to be reused. The approach taken here is more efficient in lots of ways.

We have quite a new structure for this script, beginning with a user-defined function called **init**. Not surprisingly, this is where the initial conditions are specified and variables are defined. But study the last few lines. The usual **on.paint** function is defined here as equivalent to our own user-defined **paint** function, which is then defined next. This takes care of the actual layout on the screen - including the moving ball. Finally, the timer function is defined.

Now that everything has been defined - *but nothing*

platform.apilevel = "2.0"

has actually been called - the very last function is a resize which calls **on.paint** directing it to our **init** function. So this will be run first - as required, defining and initializing everything. It then redirects any subsequent **on.paint** calls to our **paint** function. It also serves as a possible reset: any time the screen is resized or that this is called, on.paint goes back to calling the init function (just once) and things start over again.

Think about this. It means that the **init** function will only be run once, and after that calls will go to the **paint** function, where the location of the moving ball is defined and controlled. You may need to think about this for a bit - I know I did at first.

2. init: Getting Things Set Up

Top of Page

Here we will use both **require "physics"** (which provides access to the physics library) and **require "color"** (which provides access to a library of pre-defined basic colours).

Next we define the **space** in which our body will live. Within this space, we will define **gravity**. As expected, this tends to pull things downwards towards the bottom of the screen. Try different values and see the

effect - 0, 9.8, 100...

Another option for us at this stage concerns **inertia** - or

```
local W
local H
local space
local newBody
local newShape
local pos
local vel
local velX
local velY
local posX
local posY
local width
local gravity
local mass
local inertia
local elasticity
local friction
```

```
function init(gc)
    W = platform.window:width()
    H = platform.window:height()
    require "physics"
    require "color"
    space = physics.Space()
    mass = 100
    width = W/10
    gravity = 9.8
    elasticity = 1
    friction = 1
    space:setGravity(physics.Vect(0, gravity))
    inertia =
    physics.misc.momentForCircle(mass, 0,
    width, physics.Vect(0,0))
```

stage concerns **inertia** - or more formally, the moment of inertia for the circle, or whatever shape we decide for our body. The arguments for this property for a circle are the mass, the inner radius, the outer radius, and the offset of the shape from its center. Here we have set the inner radius to be 0 and the outer radius to be the width, giving us a solid ball. Think about how you might vary this and try some alternatives - for example, inner radius equal to outer radius describes just the shell of a circle - how does this behave?

The initial position is set in the same way as velocity, using a vector.

Now that we have our body defined, we can cloak it in a shape - choices are segment, box, circle and polygon. As mentioned, the shape carries attributes such as elasticity and friction. Since we are not adding these in this lesson, we could have left this step out and simply linked the body to a visible object which shares the same x and y coordinates. However, I decided to set things up as you would generally do, and to add attributes later. You will notice that the circle shape takes as its arguments the body, the radius and a value for the **offset** of this shape from its center - here we set that using a vector allowing for the radius of the circle.

Since we have gone to the trouble of creating a shape, then we may as well give it

```
        newBody = physics.Body(mass, inertia)

        newBody:setVel(physics.Vect(1000,1000))

        newBody:setPos(physics.Vect(0, 0))

        newShape =
        physics.CircleShape(newBody, width/2,
        physics.Vect(0,0))

        newShape:setRestitution(elasticity)

        newShape:setFriction(friction)

        space:addBody(newBody)

        space:addShape(newShape)

        on.paint = paint

        paint(gc)

        timer.start(0.01)

end
```

```
function paint(gc)

        pos = newBody:pos()

        vel = newBody:vel()

        velX = vel:x()

        velY = vel:y()

        posX = pos:x()

        posY = pos:y()


        if posX > W - width then

                velX = -1 * math.abs(velX)

                posX = W - width

        elseif posX < 0 then

                velX = math.abs(velX)

                posX = 0

        end

        if posY > H - width then
```

then we may as well give it some attributes - in particular, **restitution** (better thought of as **elasticity**) and **friction**. A shape with elasticity equal to 1 is perfectly elastic - any collision causes no loss of momentum. Values less than 1 are less elastic, and greater than 1 leads to some interesting behaviors - but usually to errors. Friction values greater than 0 will also result in loss of energy with each collision. Try some different values and see the effects.

Finally we add this body and shape to our space: **space:addBody(newBody)** and **space:addShape(newShape)**. To finish, redirect paint as described above, start the timer running and we are ready to set up the visuals.

```
            velY = -1 * math.abs(velY)

            posY = H - width

        elseif posY < 0 then

            velY = math.abs(velY)

            posY = 0

        end

            newBody:setPos(
            physics.Vect(posX, posY) )

            newBody:setVel(
            physics.Vect(velX, velY) )

            gc:setColorRGB(color.orange)

            gc:fillArc(posX, posY, width,
            width, 0, 360)

end
```

```
function on.timer()

    space:step(0.01)

    platform.window:invalidate()

end
```

```
function on.resize()

    on.paint = init

end
```

3. paint: Showing Off the Visuals and the Action

Top of Page

The user-defined paint function defined here is (almost) identical to the on.paint version used in the simple version. Observe the use of the **require "color"** command here - we can simply call basic colors by name - color.orange, color.yellow, etc.

Note that, while we could use any shape we liked, or even an image here, as long as we set its position to posX and posY, remember that we defined our shape formally

as a circle and, while we could attach any visual to this body at this point, the behavior physically displayed will be consistent with that of a circle.

4. Kick Starting the Show

Top of Page

The timer function is also unchanged from previously, but observe the new **on.resize** function in which the **on.paint** function is being pointed back to run the **init** function.

Placing the **on.paint = init** assignment inside the **on.resize** function does not actually bring any major advantage in this script. It would have worked just as well to have it simply sitting at the end of the script without being inside a function at all. The benefit would come if we actually wished to resize the page, since this will force the script to reset, taking account of the new window dimensions and everything would start over, in correct proportion.

Now to try some variations.

How could you make the motion pause?

Try adding an **on.enterKey** function which stops and starts the motion.

Now what about an **on.escapeKey** function that resets everything - starts it over again?

Now what about for the Player?

We have no keyboard control in the Player, so how about a button in, say, the bottom left corner which stops and starts the motion?

AND what about if, wherever I click the mouse (**on.mouseUp**) it resets the motion and starts the ball moving from that point?

These will be covered in the next lesson, and we will look at juggling multiple balls of different colors - much more fun!

Back to Top

Home ← TI-Nspire Authoring ← TI-Nspire Scripting HQ ← **Scripting Tutorial - Lesson 24**
