TI-nspire™

Scripting Tutorial - Lesson 26: (3.2) Welcome to the Physics Engine Part 4: Adding Segments as Shapes

Download supporting files for this tutorial

Download this page in PDF format

Texas Instruments TI-Nspire Scripting Support Page

Continuing once again from the previous lesson, there appears to be no visible difference once we have applied the changes described below. However, what we have done is to introduce a new type of shape - the **segment** - which joins the circle in our new repertoire of physics shapes. You will also learn some new scripting techniques which will prove useful in the future.

Bounding our Space

Top of Page

The main change in this script occurs in the **init** function. We will remove from our previous **paint** function the four statements which check when our moving bodies lie within the boundaries of the page and replace these with a different approach. The four "walls" of the screen will be defined using segments. These segments will not be visible - we will give them no shape that we can see, but

they will have a certain thickness (1 unit in this example) and they will also have restitution (elasticity) and friction attributes, which can be varied.

Can you see the benefits of such an approach?

I choose here to define a function, called **bounds**, define these segments within this function, and then simply call the function within our init routine. I choose to do this immediately after defining the **space** and before we create any other objects.

The segments will be defined within a table, here called **boundaries** and so, naturally, we begin by defining this as a local variable at the start of the script, and define the table as empty within the init function.

You will recall previously that we defined bodies and **shapes** and added these to our **space**. A slightly different approach is taken here. Since the four "walls" will not be required to move, they may be defined as **StaticShapes** instead of the usual **shape**. As such, you may notice that there is no **body** defined. Think about this.

For most physics objects, the body is the key component, the shape an optional "cloak" by which the body can be made visible and be given certain properties. For static shapes, no body is required. Instead, we define these static shapes and provide these with properties, like elasticity and friction as desired.

```
local boundaries

-----------------------------------

function init(gc)

    ...

    boundaries = {}

    ...

    space = physics.Space()

    bounds(W, H)

    ...

end

-----------------------------------

function bounds(w, h)

    -- Set the boundary walls

    for _,wall in ipairs(boundaries) do

        space:removeStaticShape(wall)

    end

    local bound

    bound = physics.SegmentShape(nil,
    physics.Vect(0,0), physics.Vect(w, 0), 1)

        : setRestitution(elasticity)

        : setFriction(friction)

    space:addStaticShape(bound)

    table.insert(boundaries, bound)

    bound = physics.SegmentShape(nil,
    physics.Vect(0,H), physics.Vect(w, h), 1)
```

We also introduce two new commands: **removeStaticShape** and **addStaticShape**.

If you study the first few lines of our bounds function, this actually clears away any existing static shapes, so that these do not accumulate with multiple calls on resize. Strictly speaking, since we are defining the table **boundaries** as empty just prior to the call to **bounds**, then this is probably not strictly necessary, but it does illustrate a neat way to run through the elements of a table using the **ipairs** command.

After defining a local variable, **bound**, we then create a segment called bound for each of the four "walls", adding each in turn to the table **boundaries**. As you see, segments are defined using the **physics.SegmentShape** command, which takes as its arguments a **body** (or, as in this case, **nil**), a vector to represent the start of the segment, another for the end point, and a value for the "**radius**" or thickness of the segment.

Note the neat technique next illustrated for adding two properties to the variable just defined. We can omit the usual syntax, bound:setRestitution(elasticity) and, since these immediately follow the definition of bound, the leading colon suffices to call this.

Each new bound is then added to the space as a static shape, and then added to the table,

```
        : setRestitution(elasticity)

        : setFriction(friction)

    space:addStaticShape(bound)

    table.insert(boundaries, bound)

    bound = physics.SegmentShape(nil,
    physics.Vect(0,0), physics.Vect(0, h), 1)

        : setRestitution(elasticity)

        : setFriction(friction)

    space:addStaticShape(bound)

    table.insert(boundaries, bound)

    bound = physics.SegmentShape(nil,
    physics.Vect(w,0), physics.Vect(w, h), 1)

        : setRestitution(elasticity)

        : setFriction(friction)

    space:addStaticShape(bound)

    table.insert(boundaries, bound)

end
```

boundaries.

The final result are four "walls" which not only reflect the balls which hit them, but actually bounce them off!!

Your challenge this time? You will notice, I am sure, that they don't quite bounce as we would expect - some of the walls seem to allow the balls to travel too far; others, too little.

Can you fix this, so that the balls bounce directly as they hit the four walls?

The next (and currently, final) lesson in the sequence will explore shapes without an infinite number of sides - in other words, we expand our shapes from circles and segments to polygons, with the option to vary the number of sides as you choose!

Back to Top

Home ← TI-Nspire Authoring ← TI-Nspire Scripting HQ ← **Scripting Tutorial - Lesson 26**
