TI-nspire™

Scripting Tutorial - Lesson 27: (3.2) Welcome to the Physics Engine Part 5: Polygons as Physics Objects

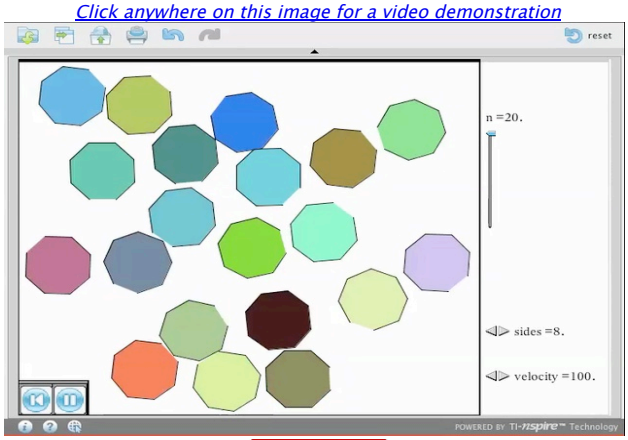
Download supporting files for this tutorial

Download this page in PDF format

Texas Instruments TI-Nspire Scripting Support Page

In the previous lesson we introduced the commands for working with segments. Here we extend our repertoire to include polygons.

Click anywhere on this image for a video demonstration



1. Bouncing and Twirling Polygons

Top of Page

The extension of our physics library to include polygons is surprisingly simple. Only two specialised commands are required to recognise our shape as polygonal: the **PolyShape** command, **physics.PolyShape(body, vertices, offset)** and the **inertia** definition: **physics.misc.momentForPoly(mass, vertices, offset)**.

- **physics.PolyShape(body, vertices, offset)**
- **physics.misc.momentForPoly(body, vertices, offset)**

Note that *Polygon shapes are bounded by a set of line segments. The enclosed area of the polygon must be **convex** and the vertices must be defined in **counterclockwise order**.*

This leads to the definition of the regular polygons used for this lesson.

You will see that the table of vertices is built from vector pairs, effectively adding x and y coordinates for each vertex in one action and producing a table that is a matrix rather than a list. Here we add an additional argument to our **table.insert** command - the number 1, which forces

```
function init(gc)
    ...
    local angleDelta = 360/nvertices
    local angle = 0
    vertices = { }
    for i=1,nvertices do
        table.insert(vertices, 1,
            physics.Vect(math.cos(math.rad(angle))*radius,
                math.sin(math.rad(angle))*radius))
        angle = angle + angleDelta
    end
    inertia = physics.misc.momentForPoly(
```

command - the number 1, which forces each vector to be added as the first element of the table. This ensures the counter-clockwise orientation required.

Of course, this list of vertex pairs is not going to be suitable for the **drawPolyLine** and **fillPolygon** commands with which we will paint our shapes. Look closely now at the code snippet which we are adding to the **paint** function.

```
                mass, vertices, physics.Vect(0,0))
        for i=1,totalBodies do
                newBody = physics.Body(mass, inertia)
                newShape = physics.PolyShape(newBody, vertices, physics.Vect(0,0))
        ...
end
```

```
function paint(gc)
        bounds(W, H)
        for _,shape in ipairs(shapes) do
                local pointcount = 1
                points = shape:points()
                for _,point in ipairs(points) do
                        vertexpoints[pointcount] = point:x()
                        vertexpoints[pointcount] = vertexpoints[pointcount] < W
                                and vertexpoints[pointcount] or W
                        vertexpoints[pointcount] = vertexpoints[pointcount] > 0
                                and vertexpoints[pointcount] or 0
                        vertexpoints[pointcount+1] = point:y()
                        vertexpoints[pointcount + 1] = vertexpoints[pointcount +
                        1] < H and vertexpoints[pointcount + 1] or H
                        vertexpoints[pointcount + 1] = vertexpoints[pointcount +
                        1] > 0 and vertexpoints[pointcount + 1] or 0
                        pointcount = pointcount + 2
                end
                gc:setColorRGB(math.random(255), math.random(255),
                math.random(255))
                gc:fillPolygon(vertexpoints)
                gc:setColorRGB(0)
                gc:drawPolyLine(vertexpoints)
        end
        ...
```

2. Painting our Polygons

Top of Page

After painting the boundary walls of our page, we begin the process of painting our polygon shapes, assigning each to the x and y coordinates of the shapes previously defined.

Our first **for** loop takes us in turn through each of the shapes in our table, **shapes**. Note again the use of the **ipairs** command with a pair of dummy variables which index each member of our table.

Assign the x-coordinate of the first point in our first shape to the first vertex and then ensure that this remains within our four walls. Do the same for the y-coordinate, and then move through each vertex point of the first shape. Then move on to each shape in turn.

What we are doing here is turning our matrix of points (**vertices**) into a list of x and y coordinates (**vertexpoints**), as required for drawing polygons.

Take a few moments to step through this process and think about what is happening here.

Finally, then, we can fill and draw our polygons in the usual way, with random colours for each.

Notice a nice little 3.2 feature - we can now use **gc:setColorRGB(** with a single number! Previously you will recall that the common usage required three numbers for Red, Green and Blue.

The rest of the **paint** command simply uses the supplied images to create our floating buttons. Subsequent functions define controls for mouse, arrow keys, enter, escape, all of which are supplemented in the example provided with sliders on a split geometry window, ensuring that the document will work optimally on both handheld and Player.

And so we reach the end of what has been, at least for me, an exciting learning journey. If you have questions, suggestions or helpful criticism, I would love to hear from you!

Back to Top
