from direct.showbase.ShowBase import ShowBase

from panda3d.core import LPoint2f, LVector2, ClockObject, LVector3f, LPlanef, LPoint3f, GeomVertexFormat, GeomVertexData, Geom, GeomTriangles, GeomVertexWriter,GeomNode, OrthographicLens, LVector3, LPoint3d, WindowProperties, loadPrcFileData

from copy import deepcopy

confVars = """

win-size 1280 720

show-frame-rate-meter True

""" #makes the window 720p and turns on the fps counter

loadPrcFileData("", confVars)

global fps, moveSpeed

fps = 90 #fps is the framerate for the physics simulation, the visuals will attempt to render at 60 fps

moveSpeed = float(100) #speed that the camera moves around

class Rectangle:

def \_\_init\_\_(self, w, h, x, z, rotation): #l, w, h are the length width and height of the box (respectively), x, y, z are the coordinates of the centre of the box, yaw, pitch, roll are the rotation of the box

self.velocity = LVector2(0, 0)

self.rotationalVelocity = 0

self.originalPoints = [LPoint2f(-w/2, -h/2), LPoint2f(w/2, -h/2), LPoint2f(w/2, h/2), LPoint2f(-w/2, h/2)]

self.points = deepcopy(self.originalPoints)#stores the current posiiotn of all points, will be updated when the box moves or rotates

self.width = w#information about the box

self.height = h

snode = GeomNode('box')#creates the geomnode (visual representation of the box)

snode.addGeom(Collision.makeQuadrilateral(self, self.points[0], self.points[1], self.points[2], self.points[3]))

#render the box to the screen

rectangleObject = render.attachNewNode(snode)

rectangleObject.setTwoSided(True)

self.rectangleModel = rectangleObject

self.rectangleModel.setPos(x, 0, z)#changes the position of the box to the inputted location

self.rectangleModel.setHpr(0, 0, rotation)

for i in range(4):

self.points[i] += LVector2(x, z) #updates all points to the new positions

def updatePosition(self, time): #call this from the physics update function, will move the box the amount specified by the velocity and rotational velocity.

self.rectangleModel.setPos(self.rectangleModel.getPos() + LVector3(self.velocity[0] \* time, 0, self.velocity[1] \* time))#updates the position of the box model (not the points)

if(self.rotationalVelocity != 0):

self.rectangleModel.setHpr(self.rectangleModel.getHpr() + LVector3f(0, 0, self.rotationalVelocity \* time))#updates the rotation of the model (not the points)

matrix = self.rectangleModel.getMat()

for i in range(4):

point = matrix.xformPoint(LPoint3f(self.originalPoints[i][0], 0, self.originalPoints[i][1]))

self.points[i] = LVector2(point[0], point[2])

def move(self, position):

self.rectangleModel.setPos(position)

global collided

collided = False

class Collision(ShowBase):

def \_\_init\_\_(self):

ShowBase.\_\_init\_\_(self)

Collision.movement(self) #enables camera control with wasd and mouse

self.scene = self.loader.loadModel("models/environment") #loads the environment, not required but makes it easier to orient yourself

self.scene.reparentTo(self.render)

self.scene.setScale(0.25, 0.25, 0.25)

self.scene.setPos(-8, 500, 0)

global rectangle0, rectangle1, rectangles, fps, clock, previousTime

rectangle0 = Rectangle(4, 10, 20, 20, 45)

rectangle0.velocity = LVector2(-5, 0)

rectangle0.rotationalVelocity = 75

rectangle1 = Rectangle(7, 3, -20, 20, 23)

rectangle1.velocity = LVector2(5, 0)

rectangle1.rotationalVelocity = -40

rectangles = [rectangle0, rectangle1]

clock = ClockObject()

previousTime = 0

self.taskMgr.doMethodLater(1/fps, self.physicsUpdate, 'physics') #every 1/fps seconds calls physicsUpdate

def physicsUpdate(self, task): #this will be called fps times per second

global clock, previousTime

frameTime = clock.get\_real\_time() - previousTime

previousTime = clock.get\_real\_time()#time since previous fram (Delta time)

rectangle0.updatePosition(frameTime)

rectangle1.updatePosition(frameTime)

global collided

if (not collided):

for rectangle in rectangles:#point-face colision detection

for rectangleCollide in rectangles:

if (rectangle != rectangleCollide):

for point in range(4):

pointPos = (LVector2(rectangleCollide.rectangleModel.getPos()[0], rectangleCollide.rectangleModel.getPos()[2]))

v = rectangle.points[point] - pointPos

x = rectangleCollide.points[1] - rectangleCollide.points[0]#x and z vectors of the rectangle

z = rectangleCollide.points[3] - rectangleCollide.points[0]

if (v.project(x).length() < (rectangleCollide.width / 2) and v.project(z).length() < (rectangleCollide.height / 2)):#if (on all three axis), the projection of the vector from the centre of the box to the point onto the edge vector < half the length of the edge vector, then the point falls within the box

rectangle.velocity = LVector3(0, 0, 0)

rectangle.rotationalVelocity = LVector3(0, 0, 0)

rectangleCollide.velocity = LVector3(0, 0, 0)

rectangleCollide.rotationalVelocity = LVector3(0, 0, 0)

print("collide - point")

collided = True

return task.again #tells the function to run again after the specified delay (1/fps seconds)

def movement(self):#enables camera and movement controls. Move the mouse to control the camera, wasd are to move forward, backwards, left right, c is to move down, space is to move up. All movements are relative to the cameras current dirction

base.disableMouse() #disables default mouse control

props = WindowProperties()

props.setCursorHidden(True) #hides the cursor

base.win.requestProperties(props)

# Setup controls.

self.keys = {} #this array will store the state of all desired keys (1 is pressed, 0 is not)

for key in ['a', 'd', 'w', 's']:

self.keys[key] = 0 #defaults key to not be pressed

self.accept(key, self.push\_key, [key, 1])#if the key is pressed

self.accept('%s-up' % key, self.push\_key, [key, 0]) #when the key is released

self.accept('escape', \_\_import\_\_('sys').exit, [0]) #closes program if escape is pressed

#Configure Camera

self.lens = OrthographicLens()

lensSize = 5

self.lens.setFilmSize(16 \* lensSize, 9 \* lensSize)

self.lens.setNear(-1000.0)

self.lens.setFar(1000.0)

self.cam.node().setLens(self.lens)

#update will update the camera position every frame

self.taskMgr.add(self.movementUpdate, 'movement')

def push\_key(self, key, value): #function to change state of keys array

self.keys[key] = value

def movementUpdate(self, task):

delta = globalClock.getDt() #time since last frame

move\_x = delta \* (moveSpeed \* self.keys['d'] - moveSpeed \* self.keys['a']) #calculates how much to move the camera on each axis

move\_z = delta \* (moveSpeed \* self.keys['w'] - moveSpeed \* self.keys['s'])

self.camera.setPos(self.camera, move\_x, 0, move\_z) #moves the camera realtive to the cameras current position and orientation

return task.cont

def makeQuadrilateral(self, point1, point2, point3, point4): #input the four points (LPoint3d) that a quadrilateral will be drawn between. Ensure that the four points make a U shape if you were to draw a line between them (Not an N or X shape)

format = GeomVertexFormat.getV3cp() #this format contains vertex location and colour of the vertex

vdata = GeomVertexData('rectangle', format, Geom.UHDynamic)

vertex = GeomVertexWriter(vdata, 'vertex')#writers for the vertex and the colour

colour = GeomVertexWriter(vdata, 'color')

for point in [point1, point2, point3, point4]:

vertex.addData3(point[0], 0, point[1]) #adds the position of the four vertexes

# adding different colors to the vertex for visibility. These colours are expressed in RGBA.

colour.addData4f(0, 0, 1, 1)

colour.addData4f(0, 0, 1, 1)

colour.addData4f(0, 0.5, 1, 1)

colour.addData4f(0.5, 0, 1, 1)

tris = GeomTriangles(Geom.UHDynamic) #creates two triangles to represent the quadrilateral

tris.addVertices(0, 1, 3)

if(point1 != point3): #if points 1 and 3 are the same, it will only generate one triangle

tris.addVertices(1, 2, 3)

rectangle = Geom(vdata)

rectangle.addPrimitive(tris)#combines the triangles into one quadrilateral

return rectangle

Collision().run()