from direct.showbase.ShowBase import ShowBase

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

from math import cos, sin, pi

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, mouseSensitivity

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

mouseSensitivity = float(50) #sensitivity for looking around

class Plane: #each face of the box has this plane object. Stores the 4 points on the box as well as the other an LPlanef object

def \_\_init\_\_ (self, point1, point2, point3, point4):

self.updatePlane(point1, point2, point3, point4)

def updatePlane(self, point1, point2, point3, point4):

self.p1 = point1

self.p2 = point2

self.p3 = point3

self.p4 = point4

self.planef = LPlanef(self.p1, self.p2, self.p3)

class Box:

def \_\_init\_\_(self, l, w, h, x, y, z, yaw, pitch, roll): #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 = LVector3(0, 0, 0)

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

self.originalPoints = [LPoint3f(-l/2, -w/2, -h/2), LPoint3f(-l/2, w/2, -h/2), LPoint3f(l/2, w/2, -h/2), LPoint3f(l/2, -w/2, -h/2), LPoint3f(-l/2, -w/2, h/2), LPoint3f(-l/2, w/2, h/2), LPoint3f(l/2, w/2, h/2), LPoint3f(l/2, -w/2, h/2)] #stores the original position of all of the points, this is used when calculating the position of the points after a rotation

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

self.pointIndex = [[5, 4, 0, 1], [7, 6, 2, 3], [6, 5, 1, 2], [4, 7, 3, 0], [6, 7, 4, 5], [1, 0, 3, 2]]#stores which points on the box correspons to each face

self.planes = [Plane(LPoint3f.zero(), LPoint3f.zero(), LPoint3f.zero(), LPoint3f.zero())] \* 6 #creates the array of planes, initializes them all to be empty

for i in range(6):

self.planes[i] = Plane(self.points[self.pointIndex[i][0]], self.points[self.pointIndex[i][1]], self.points[self.pointIndex[i][2]], self.points[self.pointIndex[i][3]]) #updates all the planes

self.length = l #information about the box

self.width = w

self.height = h

self.currentHpr = LVector3(yaw, pitch, roll)

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

for i in range(6):

snode.addGeom(Collision.makeQuadrilateral(self, self.planes[i].p1, self.planes[i].p2, self.planes[i].p3, self.planes[i].p4))

#render the box to the screen

boxObject = render.attachNewNode(snode)

boxObject.setTwoSided(True)

self.boxModel = boxObject

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

self.boxModel.setHpr(yaw, pitch, roll)

for i in range(8):

self.points[i] += LVector3(x, y, 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.currentHpr += self.rotationalVelocity \* time #time is the time since the last frame was called, stores the new rotation of the boxes

self.boxModel.setPos(self.boxModel.getPos() + self.velocity \* time)#updates the position of the box model (not the points)

self.boxModel.setHpr(self.currentHpr)#updates the rotation of the model (not the points)

h = self.currentHpr[0] / 180 \* pi

p = self.currentHpr[1] / 180 \* pi

r = self.currentHpr[2] / 180 \* pi

xdir0 = LVector3f(cos(h), sin(h), 0)

ydir0 = LVector3f(-sin(h), cos(h), 0)

zdir0 = LVector3f.unitZ()

ydir1 = ydir0 \* cos(p) + zdir0 \* sin(p)

zdir1 = zdir0 \* cos(p) - ydir0 \* sin(p)

for i in range(8):

x = self.originalPoints[i][0]

y = self.originalPoints[i][1]

z = self.originalPoints[i][2]

x1 = x \* cos(h) - y \* sin(h)

y = x \* sin(h) + y \* cos(h)

newPoint = LPoint3f(x1, y, z)

x = self.findLength(xdir0, newPoint)

y = self.findLength(ydir0, newPoint)

z = self.findLength(zdir0, newPoint)

y1 = y \* cos(p) - z \* sin(p)

z = y \* sin(p) + z \* cos(p)

newPoint = xdir0 \* x + ydir0 \* y1 + zdir0 \* z

x = self.findLength(xdir0, newPoint)

y = self.findLength(ydir1, newPoint)

z = self.findLength(zdir1, newPoint)

x1 = z \* sin(r) + x \* cos(r)

z = z \* cos(r) - x \* sin(r)

newPoint = xdir0 \* x1 + ydir1 \* y + zdir1 \* z

self.points[i] = newPoint + self.boxModel.getPos()

for i in range(6):

self.planes[i].updatePlane(self.points[self.pointIndex[i][0]], self.points[self.pointIndex[i][1]], self.points[self.pointIndex[i][2]], self.points[self.pointIndex[i][3]])

def move(self, position):

self.boxModel.setPos(position)

def findLength(self, vec1, vec2) : #returns the number of times that vec1 must be multiplied to reach vec2 after projection

vec3 = vec2.project(vec1)

if (vec1[0] != 0):

return vec3[0] / vec1[0]

elif (vec1[1] != 0):

return vec3[1] / vec1[1]

elif (vec1[2] != 0):

return vec3[2] / vec1[2]

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, 42, 0)

global box0, box1, boxes, edges, fps, clock, previousTime

box0 = Box(6, 6, 6, -20, 0.5, 20, 45, 45, 45)

box0.velocity = LVector3(5, 0, 0)

box0.rotationalVelocity = LVector3(40, 50, 80)

box1 = Box(4, 4, 5, 20, 0, 20, 45, 0, 45)

box1.velocity = LVector3(-5, 0, 0)

box1.rotationalVelocity = LVector3(20, 10, 5)

boxes = [box0, box1]

edges = [[0, 4], [1, 5], [2, 6], [3, 7], [0, 1], [3, 2], [4, 5], [7, 6], [0, 3], [1, 2], [4, 7], [5, 6]]

global boxy

boxy = Box(1, 1, 1, 0, 0, 0, 0, 0, 0)

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 fps, clock, previousTime

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

previousTime = clock.get\_real\_time()

box0.updatePosition(frameTime)

box1.updatePosition(frameTime)

xs = boxes[0].points[2][0]

ys = boxes[0].points[2][1]

zs = boxes[0].points[2][2]

global boxy

boxy.move(LPoint3f(xs, ys, zs))

global edges

global collided

if (not collided):

for box in boxes:

for boxCollide in boxes:

if (box != boxCollide):

for point in range(8):

v = box.points[point] - boxCollide.boxModel.getPos()

x = boxCollide.points[3] - boxCollide.points[0]

y = boxCollide.points[1] - boxCollide.points[0]

z = boxCollide.points[4] - boxCollide.points[0]

if (v.project(x).length() < (boxCollide.length / 2) and v.project(y).length() < (boxCollide.width / 2) and v.project(z).length() < (boxCollide.height / 2)):

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

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

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

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

print("collide - point")

collided = True

box = boxes[0]

boxCollide = boxes[1]

if (box != boxCollide):

for edgeIndex in range(12):

point = edges[edgeIndex][0]

point1 = edges[edgeIndex][1]

faceIntersection = 0

for face in range(6):

if(point != point1):

intersectionPoint = LPoint3f(0, 0, 0)

if (boxCollide.planes[face].planef.intersects\_line(intersectionPoint, box.points[point], box.points[point1])):

v = intersectionPoint - boxCollide.boxModel.getPos()

x = boxCollide.planes[face].p2 - boxCollide.planes[face].p1

y = boxCollide.planes[face].p2 - boxCollide.planes[face].p3

n = (intersectionPoint - box.points[point]).length() / (box.points[point1] - box.points[point]).length()

m = (intersectionPoint - box.points[point1]).length() / (box.points[point] - box.points[point1]).length()

if (n >= 0 and n <= 1 and m >= 0 and m <= 1):

if(v.project(x).length() < (x.length() / 2) and v.project(y).length() < (y.length() / 2)):

faceIntersection += 1

if (faceIntersection >= 2):

print("collide - edge")

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

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

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

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

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', 'c', 'space']:

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 = PerspectiveLens()

self.lens.setFov(60)

self.lens.setNear(0.01)

self.lens.setFar(1000.0)

self.heading = 0.0

self.pitch = 0.0

#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):

mw = base.mouseWatcherNode

x, y = 0, 0

if mw.hasMouse():

x, y = mw.getMouseX(), mw.getMouseY() #get the position of the mouserelative to centre

props = base.win.getProperties()

base.win.movePointer(0, props.getXSize() // 2, props.getYSize() // 2) #move the mouse back to the centre

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'])

move\_y = delta \* (moveSpeed \* self.keys['space'] - moveSpeed \* self.keys['c'])

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

self.heading -= (x \* mouseSensitivity) #how much to move the camera horizontally

if (self.pitch + y \* mouseSensitivity > 90): #clamps the vertical movement so it can't exceed straight up or straight down

self.pitch = 90

elif (self.pitch + y \* mouseSensitivity < -90):

self.pitch = -90

else:

self.pitch += (y \* mouseSensitivity) #how much to move the camera vertically

self.camera.setHpr(self.heading, self.pitch, 0)

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('square', 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], point[1], point[2]) #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)

square = Geom(vdata)

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

return square

Collision().run()