from direct.showbase.ShowBase import ShowBase

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

from math import cos, sin, pi, sqrt

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 = 120 #fps is the framerate for the physics simulation, the visuals will atempt to render at 60 fps

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

mouseSensitivity = float(50) #sensitivity for looking around

class Sphere:

def \_\_init\_\_(self, x, y, z, r, p, m): #x, y, z are the coordinates of the centre of the sphere, r is the radius, p is the percision (number of points in a semicircle). Minimum value of p is 2. m is the mass

self.velocity = LVector3(0, 0, 0) #variables that can be accessed to get properties of the sphere

self.radius = r

self.mass = m

sphere = list()#this will be a list of lists of points. Each of the lists of points will be one semicircle of the sphere.

for i in range(2 \* p):#each semi circle

semicircle = [LPoint3d(0, 0, 0)] \* (p - 1) #creates a list of points and sets them all to equal zero

for j in range(p - 1): #each point within the semi circle

ang1 = pi \* (i / p)#two component angles of position on sphere

ang2 = (pi / 2) - pi \* ((j + 1) / p)

semicircle[j] = LPoint3d((r \* cos(ang2) \* cos(ang1)), (r \* cos(ang2) \* sin(ang1)), (r \* sin(ang2)))#calculates the x, y, and z values of the point represented by the two angles

sphere.append(semicircle)#adds the semicircle of points to the sphere

snode = GeomNode('sphere')#creates an empty geomnode that will contain the sphere

for i in range(2 \* p):#creates quadrilaterals for every point on the sphere

for j in range(p - 2):

o = i + 1

if (o >= 2 \* p):

o = 0

snode.addGeom(Collision.makeQuadrilateral(self, sphere[i][j], sphere[i][j + 1], sphere[o][j + 1], sphere[o][j]))

#the top and bottom points of the sphere have not been added yet

top = LPoint3d(0, 0, r) #top point

bottom = LPoint3d(0, 0, -r) # bottom point

for i in range(2 \* p): #adds triangles betweent the top/bottom point and all neighbooring points

o = i + 1

if (o >= 2 \* p):

o = 0

snode.addGeom(Collision.makeQuadrilateral(self, top, sphere[i][0], top, sphere[o][0])) #because the first and third are both top, only a triangle will be generated

snode.addGeom(Collision.makeQuadrilateral(self, bottom, sphere[i][p - 2], bottom, sphere[o][p - 2]))

#render the sphere to the screen

sphereObject = render.attachNewNode(snode)

sphereObject.setTwoSided(True)

self.sphereModel = sphereObject

self.sphereModel.setPos(x, y, z)#changes the position of the sphere to the inputted location, Important not to generate the sphere at the desired location as the reference point will always start at (0, 0, 0). Generating the sphere at the origin then moving it after fixes this.

def setVelocity(self, velocityVector): #enables you to change the velocity

self.velocity = velocityVector

def updatePosition(self): #call this from the physics update function, will automatically move the sphere the amount specified by the velocity. If you want to add acceleration, use this same code except with acceleration/velocity instead of velocity/position

global fps

self.sphereModel.setPos(self.sphereModel.getPos() + (self.velocity / fps))

def getRadius(self): #allows you to get various properties of the sphere

return self.radius

def getVelocity(self):

return self.velocity

def getPosition(self):

return self.sphereModel.getPos()

def getMass(self):

return self.mass

class Collision(ShowBase):

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

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

lens = PerspectiveLens()

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 sphere0, sphere1

sphere0 = Sphere(20, 2, 20, 2, 8, 1)

sphere1 = Sphere(-20, 0, 22, 2, 8, 1)

sphere0.setVelocity(LVector3(-10, 0, 0))

sphere1.setVelocity(LVector3(10, 0, 0))

global fps

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

distance = self.getDistance(sphere0.getPosition(), sphere1.getPosition())

distance1 = self.getDistance(sphere0.getPosition() + (sphere0.getVelocity() / fps), sphere1.getPosition() + (sphere1.getVelocity() / fps))

collisionCheck = (distance1 < distance)

if (distance < sphere0.getRadius() + sphere1.getRadius() and collisionCheck):

pos0 = sphere0.getPosition()

pos1 = sphere1.getPosition()

vel0 = sphere0.getVelocity()

vel1 = sphere1.getVelocity()

between = pos1 - pos0

mass0 = sphere0.getMass()

mass1 = sphere1.getMass()

x0 = vel0.project(between)

x1 = vel1.project(between)

mul0 = ((mass0 - mass1) / (mass0 + mass1))

mul1 = ((2 \* mass1) / (mass0 + mass1))

mul2 = ((mass1 - mass0) / (mass0 + mass1))

fx0 = LVector3(x0[0] \* mul0, x0[1] \* mul0, x0[2] \* mul0) + LVector3(x1[0] \* mul1, x1[1] \* mul1, x1[2] \* mul1)

fx1 = LVector3(x0[0] \* mul1, x0[1] \* mul1, x0[2] \* mul1) + LVector3(x1[0] \* mul2, x1[1] \* mul2, x1[2] \* mul2)

sphere0.setVelocity(vel0 - x0 + fx0)

sphere1.setVelocity(vel1 - x1 + fx1)

sphere0.updatePosition()

sphere1.updatePosition()

return task.again

def getDistance(self, S0, S1):

between = S0 - S1

distance = sqrt(between.getX() \*\* 2 + between.getY() \*\* 2 + between.getZ() \*\* 2)

return distance

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

self.xray\_mode = False

self.show\_model\_bounds = False

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('shift-%s' % key, self.push\_key, [key, 1]) #if the key is pressed with shift, it will be registered

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.update, 'main loop')

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

self.keys[key] = value

def update(self, task):

mw = base.mouseWatcherNode

x = 0

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

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

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

#separate the coordinates from all 4 points

x1 = point1.getX()

x2 = point2.getX()

x3 = point3.getX()

x4 = point4.getX()

y1 = point1.getY()

y2 = point2.getY()

y3 = point3.getY()

y4 = point4.getY()

z1 = point1.getZ()

z2 = point2.getZ()

z3 = point3.getZ()

z4 = point4.getZ()

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

color = GeomVertexWriter(vdata, 'color')

vertex.addData3(x1, y1, z1) #adds the position of the four vertexes

vertex.addData3(x2, y2, z2)

vertex.addData3(x3, y3, z3)

vertex.addData3(x4, y4, z4)

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

color.addData4f(0, 0, 1, 1)

color.addData4f(0, 0, 1, 1)

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

color.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()