Assignment: To create a simulation of a cube being dropped onto a sloped ramp, first bouncing, then sliding into a complete stop. The forces of gravity and friction should be incorporated.

from direct.showbase.ShowBase import ShowBase

from direct.showbase.DirectObject import DirectObject

from direct.gui.DirectGui import \*

from direct.interval.IntervalGlobal import \*

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

from math import cos, pi, sqrt, sin

from array import \*

from panda3d.core import PerspectiveLens, WindowProperties

import sched, time, threading, math

from direct.task import Task

from datetime import datetime

from panda3d.core import LPlanef, LPoint3f

from panda3d.core import loadPrcFileData

confVars = """

win-size 1280 720

show-frame-rate-meter True

"""

loadPrcFileData("", confVars)

global moveSpeed

global mouseSensitivity

moveSpeed = float(1000)

mouseSensitivity = float(50)

global fps

fps = 240

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.identify = l

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)]

self.points = [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)]

self.planes = [Plane(self.points[5], self.points[4], self.points[0], self.points[1]), Plane(self.points[7], self.points[6], self.points[2], self.points[3]), Plane(self.points[6], self.points[5], self.points[1], self.points[2]), Plane(self.points[4], self.points[7], self.points[3], self.points[0]), Plane(self.points[6], self.points[7], self.points[4], self.points[5]), Plane(self.points[1], self.points[0], self.points[3], self.points[2])]

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]]

self.length = l

self.width = w

self.height = h

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

self.mass = 10

self.gravity = LVector3(0, 0, -100)

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

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

snode = GeomNode('box')#creates an empty geomnode that will contain 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, 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.

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

for i in range(8):

self.points[i] += LVector3(x, y, z)

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

global fps

self.currentHpr += self.rotationalVelocity \* time

self.boxModel.setPos(self.boxModel.getPos() + self.velocity \* time)

self.boxModel.setHpr(self.currentHpr)

for i in range(8):

x = self.originalPoints[i][0]

y = self.originalPoints[i][1]

z = self.originalPoints[i][2]

xdir = LVector3f.unitX()

ydir = LVector3f.unitY()

zdir = LVector3f.unitZ()

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

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

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

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

y1 = x \* sin(h) + y \* cos(h)

newPoint = LPoint3f(x1, y1, z)

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

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

x1 = self.findLength(xdir, newPoint)

y1 = self.findLength(ydir, newPoint)

z1 = self.findLength(zdir, newPoint)

y2 = y1 \* cos(p) - z1 \* sin(p)

z2 = y1 \* sin(p) + z1 \* cos(p)

newPoint = xdir \* x1 + ydir \* y2 + zdir \* z2

ydirOrg = ydir

xdir = xdir

ydir = ydir \* cos(p) + zdir \* sin(p)

zdir = ydirOrg \* (-sin(p)) + zdir \* cos(p)

x2 = self.findLength(xdir, newPoint)

y2 = self.findLength(ydir, newPoint)

z2 = self.findLength(zdir, newPoint)

x3 = z2 \* sin(r) + x2 \* cos(r)

z3 = z2 \* cos(r) - x2 \* sin(r)

newPoint = xdir \* x3 + ydir \* y2 + zdir \* z3

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 Plane:

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

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

def makeTriangle(x1, y1, z1, x2, y2, z2, x3, y3, z3, r, g, b, a):

format = GeomVertexFormat.getV3cp()

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

vertex = GeomVertexWriter(vdata, 'vertex')

color = GeomVertexWriter(vdata, 'color')

vertex.addData3(x1, y1, z1)

vertex.addData3(x2, y2, z2)

vertex.addData3(x3, y3, z3)

# adding different colors to the vertex for visibility

color.addData4f(r, g, b, a)

color.addData4f(r, g, b, a)

color.addData4f(r, g, b, a)

color.addData4f(r, g, b, a)

# Quads aren't directly supported by the Geom interface

# you might be interested in the CardMaker class if you are

# interested in rectangle though

tris = GeomTriangles(Geom.UHDynamic)

tris.addVertices(0, 1, 3)

tris.addVertices(1, 2, 3)

triangle = Geom(vdata)

triangle.addPrimitive(tris)

return triangle

# Note: it isn't particularly efficient to make every face as a separate Geom.

# instead, it would be better to create one Geom holding all of the faces.

x0 = 0

y0 = 0

z0 = 0

x1 = 1000

y1 = 0

z1 = 0

x2 = 0

y2 = -1000

z2 = 1000

self.cof = 2

self.angle = pi/4

# 3 points of plane

point0 = LPoint3f(x0, y0, z0)

point1 = LPoint3f(x1, y1, z1)

point2 = LPoint3f(x2, y2, z2)

self.plane0 = LPlanef(point0, point1, point2)

self.normal = self.plane0.getNormal()

print("normal = ", self.normal)

planeTriangle = makeTriangle(point0.getX(), point0.getY(), point0.getZ(), point1.getX(), point1.getY(), point1.getZ(), point2.getX(), point2.getY(), point2.getZ(), 1, 1, 1, 1)

snode = GeomNode('square')

snode.addGeom(planeTriangle)

plane = render.attachNewNode(snode)

plane.setTwoSided(True)

self.planeModel = plane

def disToPlane(self, point):

return self.plane0.distToPlane(point)

def getClosePoint(self, point):

return self.plane0.project(point)

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 Collision(ShowBase):

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

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

lens = PerspectiveLens()

base.setFrameRateMeter(True)

Collision.movement(self)

self.scene = self.loader.loadModel("models/environment")

self.scene.reparentTo(self.render)

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

self.scene.setPos(-8, 42, 0)

global t, cube0, plane0, fps, timeSinCol, checkBounce

#plane collision sphere

plane0 = Plane((0,0,0),(0,0,0),(0,0,0),(0,0,0))

cube0 = Box(30, 30, 30, 100, -600, 1000, 0, 45, 0)

cube0.velocity = LVector3(0, 0, 0)

timeSinCol = 10000

checkBounce = True

self.taskMgr.doMethodLater(1/fps, self.physicsUpdate, 'physics')

def physicsUpdate(self, task):

global timeSinCol, projNor, checkBounce

projNor = (0,0,0)

cube0.acceration = LVector3(0,0,0)

fn = 0

ff = 0

colPoints = []

timeSinCol += 1

for x in cube0.points:

disToPlane = plane0.disToPlane(x)

if (disToPlane > -0.5 and disToPlane < 0.5):

colPoints.append(x)

if(len(colPoints) >= 3):

if (cube0.velocity.project(plane0.normal).length() < 1):

cube0.velocity -= cube0.velocity.project(plane0.normal)

# Fn = -cos(45)(Fg)

# Fn = -cos(45)(m)(g)

fn = -plane0.normal.normalized() \* cube0.mass \* cube0.gravity.length() \* math.cos(plane0.angle)

# #ff = cof(fn)

ff = -cube0.velocity.normalized() \* fn.length() \* plane0.cof

elif(timeSinCol>1):

timeSinCol = 0

print("collided")

projNormal = -cube0.velocity.project(plane0.normal)

#print(projNormal)

planeVec = plane0.getClosePoint(x + cube0.velocity) - x

projPlane = cube0.velocity.project(planeVec)

cube0.velocity = projNormal/3 + projPlane

# Fn = -cos(45)(Fg)

# Fn = -cos(45)(m)(g)

fn = -plane0.normal.normalized() \* cube0.mass \* cube0.gravity.length() \* math.cos(plane0.angle)

# #ff = cof(fn)

ff = -cube0.velocity.normalized() \* fn.length() \* plane0.cof

# projNor = cube0.velocity.project(plane0.normal)

#a = g + fn/m + ff/m

cube0.acceration = cube0.acceration + cube0.gravity + fn/cube0.mass + ff/cube0.mass

#v = v + at

cube0.velocity = cube0.velocity + (cube0.acceration \* 1/fps)

print(cube0.velocity)

print(task.time)

cube0.updatePosition(task.time + 1/fps)

return task.again

def movement(self):

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 = {}

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

self.keys[key] = 0 #array that stores the state of the above keys (1 is pressed down, 0 is not)

self.accept(key, self.push\_key, [key, 1])

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

self.accept('%s-up' % key, self.push\_key, [key, 0])

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

# Setup camera

self.lens = PerspectiveLens()

self.lens.setFov(60)

self.lens.setNear(0.01)

self.lens.setFar(1000.0)

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

self.heading = 0.0

self.pitch = 0.0

self.taskMgr.add(self.update, 'main loop')

def push\_key(self, key, value):

self.keys[key] = value

def update(self, task):

mw = base.mouseWatcherNode

x = 0

y = 0

if mw.hasMouse():

# get the position relative to centre

x, y = mw.getMouseX(), mw.getMouseY()

# move mouse back to center

props = base.win.getProperties()

base.win.movePointer(0, props.getXSize() // 2, props.getYSize() // 2)

delta = globalClock.getDt()

move\_x = delta \* moveSpeed \* self.keys['d'] - delta \* moveSpeed \* self.keys['a']

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)

self.heading += (-x \* mouseSensitivity)

if (self.pitch + y \* mouseSensitivity > 90):

self.pitch = 90

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

self.pitch = -90

else:

self.pitch += (y \* mouseSensitivity)

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

t = Collision()

t.run()