from panda3d.egg import \*

from panda3d.core import \*

from copy import deepcopy

import numpy as np

import cv2

import copy

from pandac.PandaModules import \*

#function to find distance between two coordinates

def distancecoordinates(a, b):

x = pow(a[0] - b[0], 2)

y = pow(a[1] - b[1], 2)

Distance = pow(x+y, 0.5)

return Distance

#function to find wall attributes

def calcwallfease(line, wallw = -1):

a = abs(line[0][0] - line[1][0])

b = abs(line[0][1] - line[1][1])

if a>b:

if wallw<0 or b<=wallw:

return 0

elif a<b:

if wallw< 0 or a<wallw:

return 1

else:

return -1

#Class used to define objects without models

class MaterialtoOBJ:

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

self.filename = None

self.name = "default"

self.dtexture = None

self.dmaterial = None

self.attrib = {}

self.attrib["d"] = 1.0

self.attrib["illum"] = 2

self.attrib["Kd"] = [1.0, 1.0, 1.0]

self.attrib["Ka"] = [0.0, 0.0, 0.0]

self.attrib["Ks"] = [0.0, 0.0, 0.0]

self.attrib["Ke"] = [0.0, 0.0, 0.0]

def getkeyval(self, key):

if key in self.attrib:

return self.attrib[key]

return None

def gettexture(self):

if self.dtexture:

return self.dtexture

key1 = "MaterialKvals"

boolreq = False

if key1 in self.attrib:

boolreq = True

if not boolreq:

return None

texture = str(self.name) + "\_diffuse"

Material = EggTexture(texture, self.getkeyval(key1))

Material.setFormat(EggTexture.FRgb)

Material.setMagfilter(EggTexture.FTLinearMipmapLinear)

Material.setMinfilter(EggTexture.FTLinearMipmapLinear)

Material.setWrapU(EggTexture.WMRepeat)

Material.setWrapV(EggTexture.WMRepeat)

self.dtexture = Material

return self.dtexture

def getmaterial(self):

if self.dmaterial:

return self.dmaterial

Material = EggMaterial(self.name + "\_mat")

rgb = self.getkeyval("Kd")

if rgb is not None:

Material.setDiff(Vec4(rgb[0], rgb[1], rgb[2], 1.0))

rgb = self.getkeyval("Ka")

if rgb is not None:

Material.setAmb(Vec4(rgb[0], rgb[1], rgb[2], 1.0))

rgb = self.getkeyval("Ks")

if rgb is not None:

Material.setSpec(Vec4(rgb[0], rgb[1], rgb[2], 1.0))

self.dmaterial = Material

return self.dmaterial

def createnewkeyval(self, key, value):

self.attrib[key] = value

return self

#class which creates the simulation

class Blueprint():

#Initialisation function to load in the materials for the floor,ceiling,wall,and door

#As well as the egg files of the various models

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

self.wallw = 0.002

self.wallh = 0.3

self.doorw = self.wallw

self.doorh = self.wallh \* 0.8

self.filename = filename

self.floormaterial = MaterialtoOBJ()

self.floormaterial.name = 'floor'

self.floormaterial.createnewkeyval('MaterialKvals', 'Models/floor.jpg')

self.ceilingMat = MaterialtoOBJ()

self.ceilingMat.name = 'ceiling'

self.ceilingMat.createnewkeyval('MaterialKvals', 'Models/ceiling.jpg')

self.wallmaterial = MaterialtoOBJ()

self.wallmaterial.name = 'wall\_1'

self.wallmaterial.createnewkeyval('MaterialKvals', 'Models/wall.jpg')

self.doormaterial = MaterialtoOBJ()

self.doormaterial.name = 'door'

self.doormaterial.createnewkeyval('MaterialKvals', 'Models/door.jpg')

self.elementNodes = {}

self.elementNodes['bed'] = base.loader.loadModel('Models/tinker.egg')

self.elementNodes['bedv'] = base.loader.loadModel('Models/bedv.egg')

self.elementNodes['ward'] = base.loader.loadModel('Models/ward.egg')

self.elementNodes['wardv'] = base.loader.loadModel('Models/wardv.egg')

self.elementNodes['desk'] = base.loader.loadModel('Models/table.egg')

self.elementNodes['deskv'] = base.loader.loadModel('Models/table2.egg')

return

#Reading the text file generated by the GUI and splitting the sections

def read(self):

guifile = open('Models/floorplan\_1.txt', 'r')

self.walls = []

self.doors = []

self.elements = []

for i in guifile.readlines():

i = i.strip()

values = i.split('\t')

#Reading Dimensions

if len(values) == 2:

self.width = float(values[0])

self.height = float(values[1])

self.maxDim = max(self.width, self.height)

#Reading walls

elif len(values) == 6:

wall = []

print("Wall Found")

for i in range(4):

wall.append(float(values[i]))

continue

wallDim = calcwallfease(((wall[0], wall[1]), (wall[2], wall[3])))

wall[wallDim], wall[2 + wallDim] = min(wall[wallDim], wall[2 + wallDim]), max(wall[wallDim], wall[2 + wallDim])

wall[1 - wallDim] = wall[3 - wallDim] = (wall[1 - wallDim] + wall[3 - wallDim]) / 2

wall.append(int(values[4]) - 1)

wall.append(int(values[5]) - 1)

for j in range(2):

wall[j \* 2 + 0] /= self.maxDim

wall[j \* 2 + 1] /= self.maxDim

continue

self.walls.append(wall)

#Reading Objects

elif len(values) == 7:

print("Object Found")

item = []

for k in range(4):

item.append(float(values[k]))

for j in range(2):

item[j \* 2 + 0] /= self.maxDim

item[j \* 2 + 1] /= self.maxDim

continue

if values[4] == 'door':

self.doors.append(item)

else:

item.append(values[4])

self.elements.append(item)

continue

return

#Function to simulate floor using the restrictions of walls

#Also simulates ceiling

def sFloor(self, data):

floorvectors = EggGroup('floor')

data.addChild(floorvectors)

VertexList = EggVertexPool('floor\_vertex')

floorvectors.addChild(VertexList)

roomlimits = []

for limit in self.walls:

if limit[4] == 10 or limit[5] == 10:

roomlimits.append(copy.deepcopy(limit))

continue

exteriorOpenings = []

for limit in roomlimits:

wallDim = calcwallfease((limit[:2], limit[2:4]))

for doori, door in enumerate(self.doors):

if calcwallfease((door[:2], door[2:4])) != wallDim:

continue

if door[wallDim] >= limit[wallDim] and door[2 + wallDim] <= limit[2 + wallDim] and abs(door[1 - wallDim] - limit[1 - wallDim]) <= self.wallw:

exteriorOpenings.append(doori)

continue

continue

minDistance = 10000

mainDoorind = -1

for element in self.elements:

if element[4] == 'entrance':

for doori in exteriorOpenings:

door = self.doors[doori]

distance = pow(pow((door[0] + door[2]) / 2 - (element[0] + element[2]) / 2, 2) + pow((door[1] + door[3]) / 2 - (element[1] + element[3]) / 2, 2), 0.5)

if distance < minDistance:

minDistance = distance

mainDoorind = doori

continue

break

continue

self.startCameraPos = [0.5, -0.5, self.wallh \* 0.5]

self.startTarget = [0.5, 0.5, self.wallh \* 0.5]

if mainDoorind >= 0:

mainDoor = self.doors[mainDoorind]

wallDim = calcwallfease((mainDoor[:2], mainDoor[2:4]))

fixedValue = (mainDoor[1 - wallDim] + mainDoor[3 - wallDim]) / 2

imageSize = [self.width / self.maxDim, self.height / self.maxDim]

side = int(fixedValue < imageSize[1 - wallDim] \* 0.5) \* 2 - 1

self.startCameraPos[wallDim] = (mainDoor[wallDim] + mainDoor[2 + wallDim]) / 2

self.startTarget[wallDim] = (mainDoor[wallDim] + mainDoor[2 + wallDim]) / 2

self.startCameraPos[1 - wallDim] = fixedValue - 0.5 \* side

self.startTarget[1 - wallDim] = fixedValue + 0.5 \* side

self.startCameraPos[0] = 1 - self.startCameraPos[0]

self.startTarget[0] = 1 - self.startTarget[0]

walllimitationss = []

visitedMask = {}

gap = 5.0 / self.maxDim

for ind, limit in enumerate(roomlimits):

if ind in visitedMask:

continue

visitedMask[ind] = True

walllimitations = []

walllimitations.append(limit)

for loopWall in walllimitations:

for nextwind, nextw in enumerate(roomlimits):

if nextwind in visitedMask:

continue

if distancecoordinates(nextw[:2], loopWall[2:4]) < gap:

walllimitations.append(nextw)

visitedMask[nextwind] = True

break

elif distancecoordinates(nextw[2:4], loopWall[2:4]) < gap:

nextw[0], nextw[2] = nextw[2], nextw[0]

nextw[1], nextw[3] = nextw[3], nextw[1]

walllimitations.append(nextw)

visitedMask[nextwind] = True

break

continue

continue

walllimitationss.append(walllimitations)

continue

for walllimitations in walllimitationss:

floorshape = EggPolygon()

floorvectors.addChild(floorshape)

floorshape.setTexture(self.floormaterial.gettexture())

floorshape.setMaterial(self.floormaterial.getmaterial())

for ind, limit in enumerate(walllimitations):

if ind == 0:

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - limit[0], limit[1], 0))

Vertexobj.setUv(Point2D(limit[0] \* self.maxDim / self.width, 1 - limit[1] \* self.maxDim / self.height))

floorshape.addVertex(VertexList.addVertex(Vertexobj))

else:

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - (limit[0] + walllimitations[ind - 1][2]) / 2, (limit[1] + walllimitations[ind - 1][3]) / 2, 0))

Vertexobj.setUv(Point2D((limit[0] + walllimitations[ind - 1][2]) / 2 \* self.maxDim / self.width, 1 - (limit[1] + walllimitations[ind - 1][3]) / 2 \* self.maxDim / self.height))

floorshape.addVertex(VertexList.addVertex(Vertexobj))

if ind == len(walllimitations) - 1:

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - limit[2], limit[3], 0))

Vertexobj.setUv(Point2D(limit[2] \* self.maxDim / self.width, 1 - limit[3] \* self.maxDim / self.height))

floorshape.addVertex(VertexList.addVertex(Vertexobj))

continue

continue

ceilingvals = EggGroup('ceiling')

data.addChild(ceilingvals)

VertexList = EggVertexPool('ceiling\_vertex')

ceilingvals.addChild(VertexList)

for walllimitations in walllimitationss:

floorshape = EggPolygon()

ceilingvals.addChild(floorshape)

floorshape.setTexture(self.ceilingMat.gettexture())

floorshape.setMaterial(self.ceilingMat.getmaterial())

for ind, limit in enumerate(walllimitations):

if ind == 0:

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - limit[0], limit[1], self.wallh))

Vertexobj.setUv(Point2D(limit[0], 1 - limit[1]))

floorshape.addVertex(VertexList.addVertex(Vertexobj))

else:

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - (limit[0] + walllimitations[ind - 1][2]) / 2, (limit[1] + walllimitations[ind - 1][3]) / 2, self.wallh))

Vertexobj.setUv(Point2D((limit[0] + walllimitations[ind - 1][2]) / 2, 1 - (limit[1] + walllimitations[ind - 1][3]) / 2))

floorshape.addVertex(VertexList.addVertex(Vertexobj))

if ind == len(walllimitations) - 1:

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - limit[2], limit[3], self.wallh))

Vertexobj.setUv(Point2D(limit[2], 1 - limit[3]))

floorshape.addVertex(VertexList.addVertex(Vertexobj))

continue

continue

return

# Function to place walls

def sWalls(self, data):

wallsvals = EggGroup('walls')

data.addChild(wallsvals)

VertexList = EggVertexPool('wall\_vertex')

data.addChild(VertexList)

for ind, wall in enumerate(self.walls):

wallvals = EggGroup('wall')

wallsvals.addChild(wallvals)

wallDim = calcwallfease((wall[:2], wall[2:4]))

if wallDim == 0:

widthvalues = (0, self.wallw)

else:

widthvalues = (self.wallw, 0)

wallshape = EggPolygon()

wallvals.addChild(wallshape)

wallshape.setTexture(self.wallmaterial.gettexture())

wallshape.setMaterial(self.wallmaterial.getmaterial())

values = [wall[wallDim] - self.wallw + 0.0001, wall[2 + wallDim] + self.wallw - 0.0001]

for door in self.doors:

if calcwallfease((door[:2], door[2:4])) != wallDim:

continue

if door[wallDim] >= wall[wallDim] and door[2 + wallDim] <= wall[2 + wallDim] and abs(door[1 - wallDim] - wall[1 - wallDim]) <= self.wallw:

values.append(door[wallDim])

values.append(door[2 + wallDim])

continue

values.sort()

fixedValue = (wall[1 - wallDim] + wall[3 - wallDim]) / 2

for valueind, value in enumerate(values):

if valueind % 2 == 0 and valueind > 0:

Vertexobj = EggVertex()

if wallDim == 0:

Vertexobj.setPos(Point3D(1 - (value - widthvalues[0]), fixedValue - widthvalues[1], self.doorh))

else:

Vertexobj.setPos(Point3D(1 - (fixedValue - widthvalues[0]), value - widthvalues[1], self.doorh))

Vertexobj.setUv(Point2D(self.doorh / self.wallh, (value - wall[wallDim]) / (wall[2 + wallDim] - wall[wallDim])))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

if wallDim == 0:

Vertexobj.setPos(Point3D(1 - (value - widthvalues[0]), fixedValue - widthvalues[1], 0))

else:

Vertexobj.setPos(Point3D(1 - (fixedValue - widthvalues[0]), value - widthvalues[1], 0))

Vertexobj.setUv(Point2D(0, (value - wall[wallDim]) / (wall[2 + wallDim] - wall[wallDim])))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

if valueind % 2 == 1 and valueind + 1 < len(values):

Vertexobj = EggVertex()

if wallDim == 0:

Vertexobj.setPos(Point3D(1 - (value - widthvalues[0]), fixedValue - widthvalues[1], self.doorh))

else:

Vertexobj.setPos(Point3D(1 - (fixedValue - widthvalues[0]), value - widthvalues[1], self.doorh))

Vertexobj.setUv(Point2D(self.doorh / self.wallh, (value - wall[wallDim]) / (wall[2 + wallDim] - wall[wallDim])))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

continue

Vertexobj = EggVertex()

if wallDim == 0:

Vertexobj.setPos(Point3D(1 - (values[len(values) - 1] - widthvalues[0]), fixedValue - widthvalues[1], self.wallh))

else:

Vertexobj.setPos(Point3D(1 - (fixedValue - widthvalues[0]), values[len(values) - 1] - widthvalues[1], self.wallh))

Vertexobj.setUv(Point2D(1, 1))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

if wallDim == 0:

Vertexobj.setPos(Point3D(1 - (values[0] - widthvalues[0]), fixedValue - widthvalues[1], self.wallh))

else:

Vertexobj.setPos(Point3D(1 - (fixedValue - widthvalues[0]), values[0] - widthvalues[1], self.wallh))

Vertexobj.setUv(Point2D(1, 0))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

wallshape = EggPolygon()

wallvals.addChild(wallshape)

wallshape.setTexture(self.wallmaterial.gettexture())

wallshape.setMaterial(self.wallmaterial.getmaterial())

#widthvalues = (0.1, 0.1)

for valueind, value in enumerate(values):

if valueind % 2 == 0 and valueind > 0:

Vertexobj = EggVertex()

if wallDim == 0:

Vertexobj.setPos(Point3D(1 - (value + widthvalues[0]), fixedValue + widthvalues[1], self.doorh))

else:

Vertexobj.setPos(Point3D(1 - (fixedValue + widthvalues[0]), value + widthvalues[1], self.doorh))

Vertexobj.setUv(Point2D(self.doorh / self.wallh, (value - wall[wallDim]) / (wall[2 + wallDim] - wall[wallDim])))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

if wallDim == 0:

Vertexobj.setPos(Point3D(1 - (value + widthvalues[0]), fixedValue + widthvalues[1], 0))

else:

Vertexobj.setPos(Point3D(1 - (fixedValue + widthvalues[0]), value + widthvalues[1], 0))

Vertexobj.setUv(Point2D(0, (value - wall[wallDim]) / (wall[2 + wallDim] - wall[wallDim])))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

if valueind % 2 == 1 and valueind + 1 < len(values):

Vertexobj = EggVertex()

if wallDim == 0:

Vertexobj.setPos(Point3D(1 - (value + widthvalues[0]), fixedValue + widthvalues[1], self.doorh))

else:

Vertexobj.setPos(Point3D(1 - (fixedValue + widthvalues[0]), value + widthvalues[1], self.doorh))

Vertexobj.setUv(Point2D(self.doorh / self.wallh, (value - wall[wallDim]) / (wall[2 + wallDim] - wall[wallDim])))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

continue

Vertexobj = EggVertex()

if wallDim == 0:

Vertexobj.setPos(Point3D(1 - (values[len(values) - 1] + widthvalues[0]), fixedValue + widthvalues[1], self.wallh))

else:

Vertexobj.setPos(Point3D(1 - (fixedValue + widthvalues[0]), values[len(values) - 1] + widthvalues[1], self.wallh))

Vertexobj.setUv(Point2D(1, 1))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

if wallDim == 0:

Vertexobj.setPos(Point3D(1 - (values[0] + widthvalues[0]), fixedValue + widthvalues[1], self.wallh))

else:

Vertexobj.setPos(Point3D(1 - (fixedValue + widthvalues[0]), values[0] + widthvalues[1], self.wallh))

Vertexobj.setUv(Point2D(1, 0))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

wallshape = EggPolygon()

wallvals.addChild(wallshape)

wallshape.setTexture(self.wallmaterial.gettexture())

wallshape.setMaterial(self.wallmaterial.getmaterial())

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - values[0], fixedValue - widthvalues[1], 0))

Vertexobj.setUv(Point2D(0, 0))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - values[0], fixedValue - widthvalues[1], self.wallh))

Vertexobj.setUv(Point2D(0, 1))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - values[0], fixedValue + widthvalues[1], self.wallh))

Vertexobj.setUv(Point2D(1, 1))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - values[0], fixedValue + widthvalues[1], 0))

Vertexobj.setUv(Point2D(1, 0))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

wallshape = EggPolygon()

wallvals.addChild(wallshape)

wallshape.setTexture(self.wallmaterial.gettexture())

wallshape.setMaterial(self.wallmaterial.getmaterial())

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - values[0], fixedValue - widthvalues[1], self.wallh))

Vertexobj.setUv(Point2D(0, 0))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - values[len(values) - 1], fixedValue - widthvalues[1], self.wallh))

Vertexobj.setUv(Point2D(0, 1))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - values[len(values) - 1], fixedValue + widthvalues[1], self.wallh))

Vertexobj.setUv(Point2D(1, 1))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - values[0], fixedValue + widthvalues[1], self.wallh))

Vertexobj.setUv(Point2D(1, 0))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

wallshape = EggPolygon()

wallvals.addChild(wallshape)

wallshape.setTexture(self.wallmaterial.gettexture())

wallshape.setMaterial(self.wallmaterial.getmaterial())

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - values[len(values) - 1], fixedValue - widthvalues[1], self.wallh))

Vertexobj.setUv(Point2D(0, 0))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - values[len(values) - 1], fixedValue - widthvalues[1], 0))

Vertexobj.setUv(Point2D(0, 1))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - values[len(values) - 1], fixedValue + widthvalues[1], 0))

Vertexobj.setUv(Point2D(1, 1))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - values[len(values) - 1], fixedValue + widthvalues[1], self.wallh))

Vertexobj.setUv(Point2D(1, 0))

wallshape.addVertex(VertexList.addVertex(Vertexobj))

return

#Function to Generate Doors

def sDoors(self, data):

doorsvals = EggGroup('doors')

data.addChild(doorsvals)

VertexList = EggVertexPool('door\_vertex')

doorsvals.addChild(VertexList)

for doori, door in enumerate(self.doors):

doorvals = EggGroup('door\_' + str(doori))

doorsvals.addChild(doorvals)

wallDim = calcwallfease((door[:2], door[2:4]))

widthvalues = (self.doorw, 0)

doorshape = EggPolygon()

doorvals.addChild(doorshape)

doorshape.setTexture(self.doormaterial.gettexture())

doorshape.setMaterial(self.doormaterial.getmaterial())

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - (door[0] - widthvalues[0]), door[1] - widthvalues[1], 0))

Vertexobj.setUv(Point2D(0, 0))

doorshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - (door[2] - widthvalues[0]), door[3] - widthvalues[1], 0))

Vertexobj.setUv(Point2D(1, 0))

doorshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - (door[2] - widthvalues[0]), door[3] - widthvalues[1], self.doorh))

Vertexobj.setUv(Point2D(1, 1))

doorshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - (door[0] - widthvalues[0]), door[1] - widthvalues[1], self.doorh))

Vertexobj.setUv(Point2D(0, 1))

doorshape.addVertex(VertexList.addVertex(Vertexobj))

doorshape = EggPolygon()

doorvals.addChild(doorshape)

doorshape.setTexture(self.doormaterial.gettexture())

doorshape.setMaterial(self.doormaterial.getmaterial())

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - (door[0] + widthvalues[0]), door[1] + widthvalues[1], 0))

Vertexobj.setUv(Point2D(0, 0))

doorshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - (door[2] + widthvalues[0]), door[3] + widthvalues[1], 0))

Vertexobj.setUv(Point2D(1, 0))

doorshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - (door[2] + widthvalues[0]), door[3] + widthvalues[1], self.doorh))

Vertexobj.setUv(Point2D(1, 1))

doorshape.addVertex(VertexList.addVertex(Vertexobj))

Vertexobj = EggVertex()

Vertexobj.setPos(Point3D(1 - (door[0] + widthvalues[0]), door[1] + widthvalues[1], self.doorh))

Vertexobj.setUv(Point2D(0, 1))

doorshape.addVertex(VertexList.addVertex(Vertexobj))

continue

return

def sceneElements(self, scene):

#Iterate over 3D models wanted

for element in self.elements:

#Check if model is available

if element[4] not in self.elementNodes:

continue

#Create copy of model to work with

model = deepcopy(self.elementNodes[element[4]])

#Tinkercad models were rotated 90 degrees so this corrects them

model.setHpr(0, -90, 0)

# model.setHpr(0, 0, 0)

#Get bounds

minimumval, maximumval = model.getTightBounds()

dimensions = Point3(maximumval - minimumval)

minDistances = [self.maxDim, self.maxDim, self.maxDim, self.maxDim]

for wall in self.walls:

wallDim = calcwallfease(((wall[0], wall[1]), (wall[2], wall[3])))

if wallDim == -1:

continue

if ((element[wallDim] + element[2 + wallDim]) / 2 - wall[wallDim]) \* ((element[wallDim] + element[2 + wallDim]) / 2 - wall[2 + wallDim]) > 0:

continue

side = int(wall[1 - wallDim] > (element[1 - wallDim] + element[3 - wallDim]) / 2)

ind = wallDim \* 2 + side

distance = abs(wall[1 - wallDim] - element[1 - wallDim + side \* 2])

if distance < minDistances[ind]:

minDistances[ind] = distance

continue

orientation = 0

#Scale Object

scaleX = (element[2] - element[0]) / dimensions.getX()

scaleY = (element[3] - element[1]) / dimensions.getY()

scaleZ = max(scaleX, scaleY)

model.setScale(scaleX, scaleY, scaleZ)

#Place Object

model.setPos(1 - element[0] - maximumval.getX() \* scaleX, element[1] - minimumval.getY() \* scaleY, -minimumval.getZ() \* scaleZ)

model.setTwoSided(True)

#AttachModel to Scene

model.reparentTo(scene)

continue

return

def generateSimulation(self):

#Create data element for simulation

data = EggData()

#create simulation environment/group

simulation = EggGroup('model')

#add environment to data

data.addChild(simulation)

#create floors

self.sFloor(simulation)

#create walls

self.sWalls(simulation)

#create doorways

self.sDoors(simulation)

#create scene area for elements

inside = NodePath(loadEggData(data))

#create specified objects

self.sceneElements(inside)

return inside