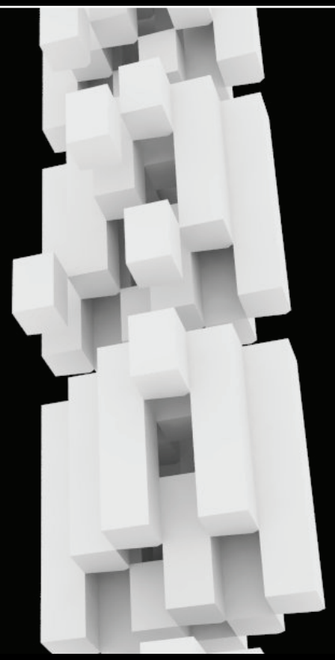
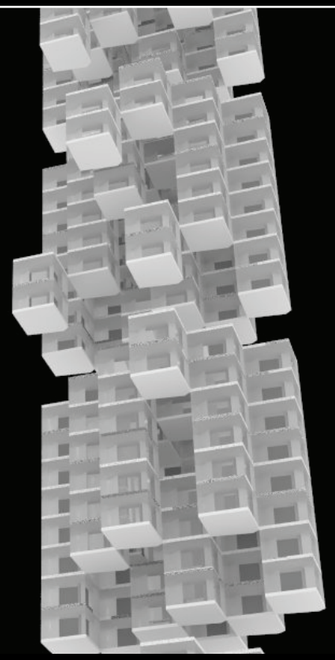
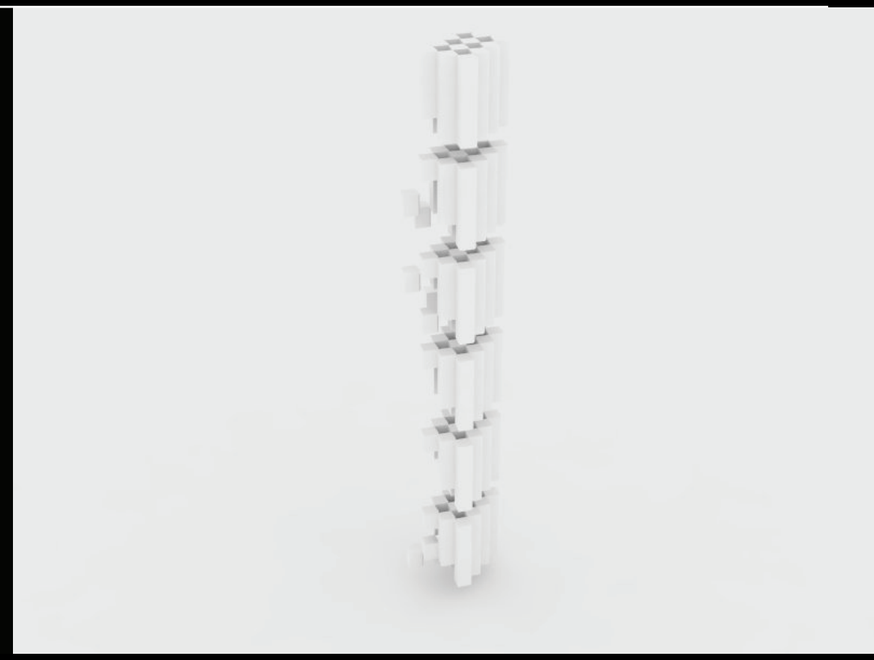
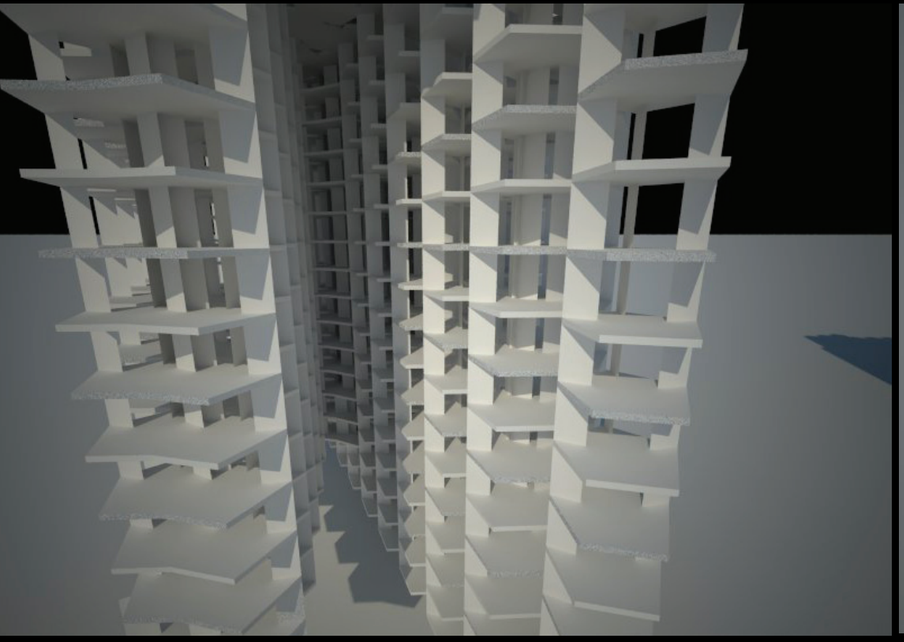
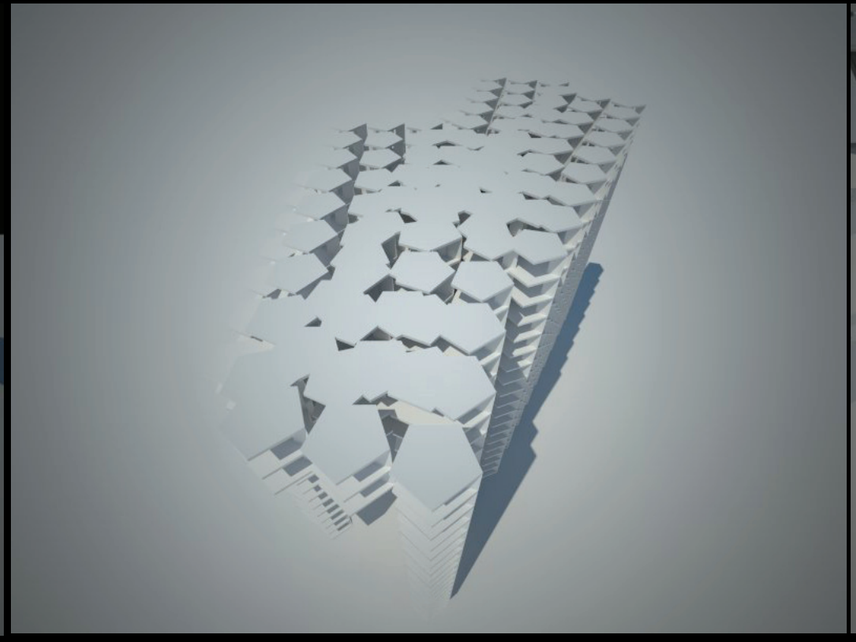


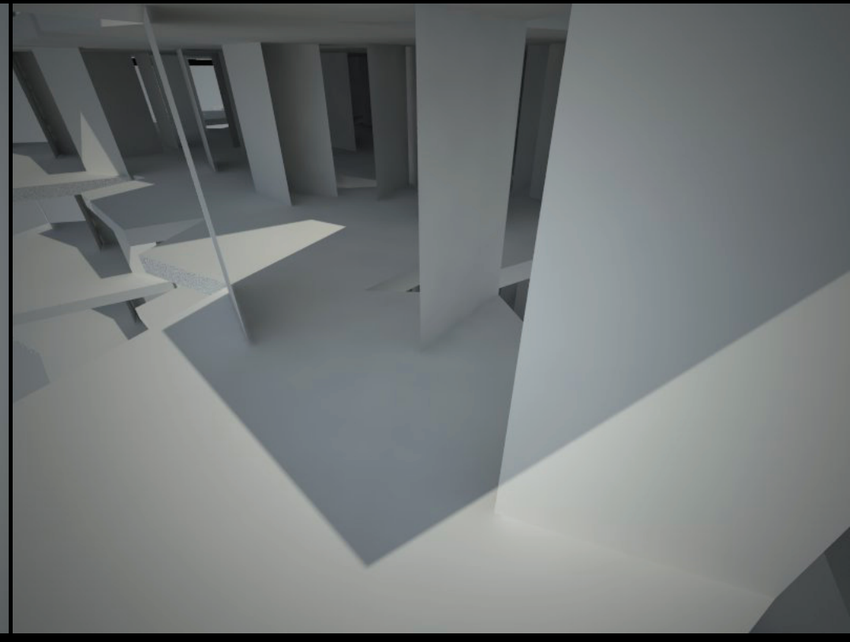




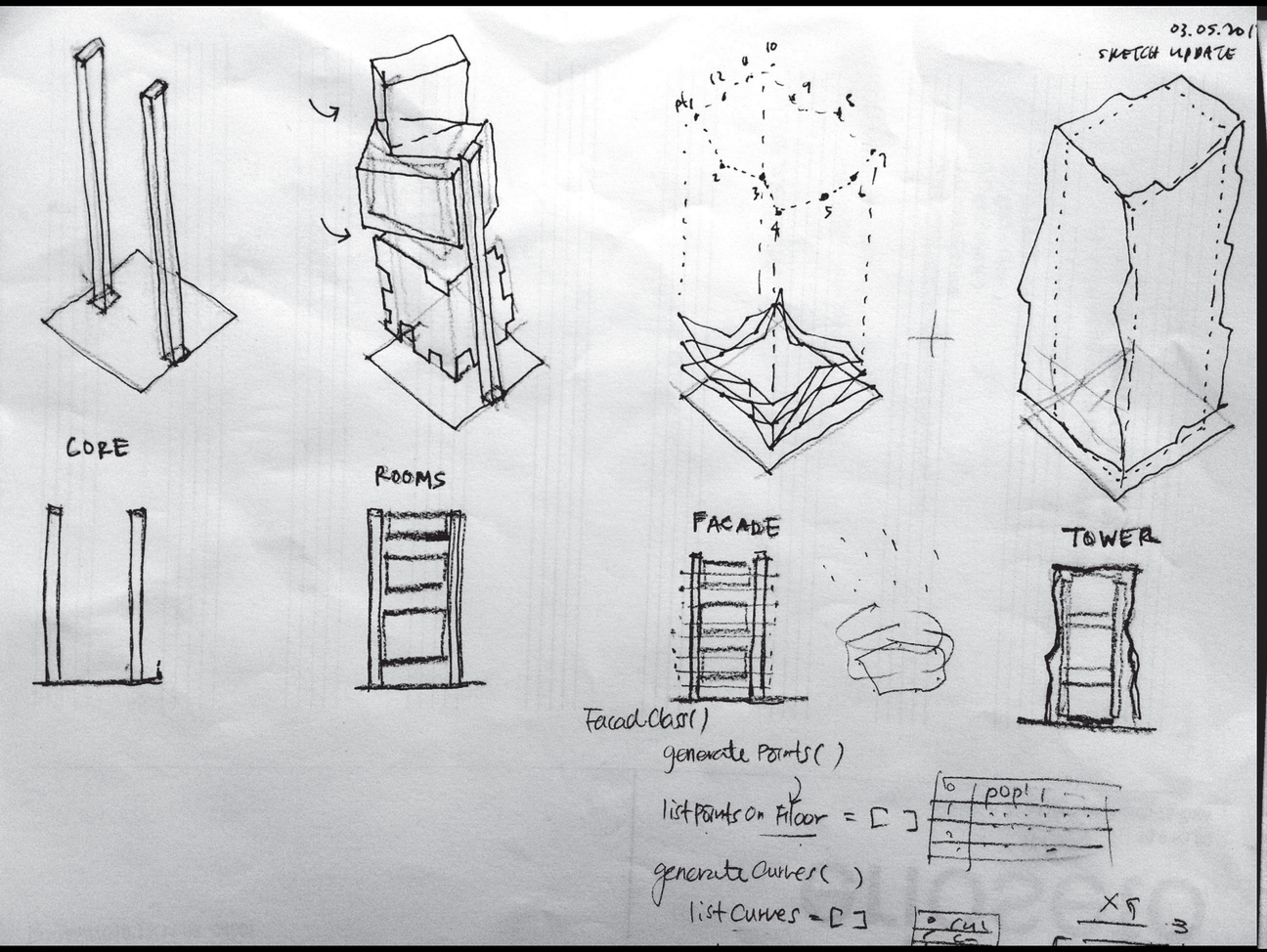
REGULAR VS RANDOM

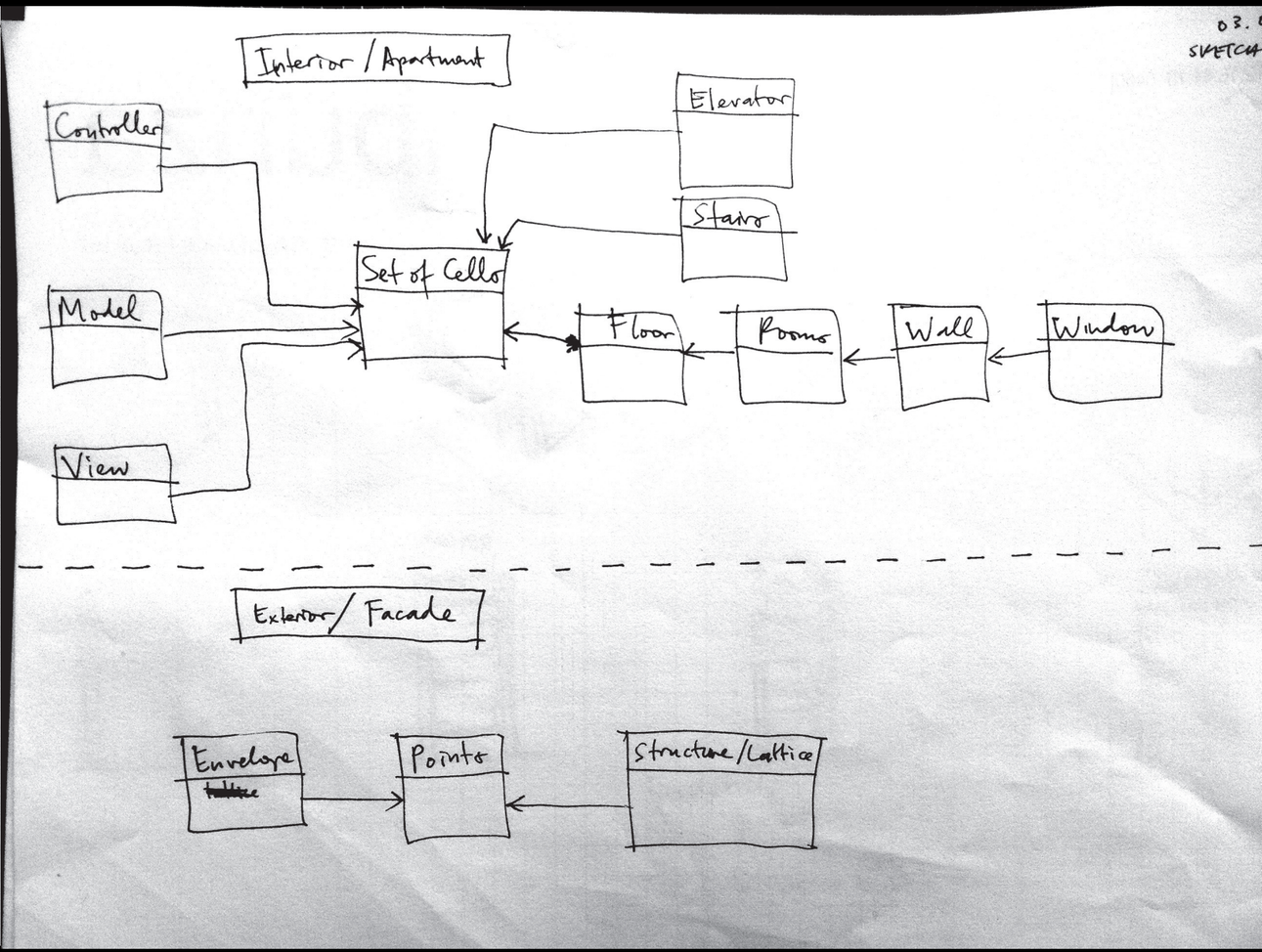




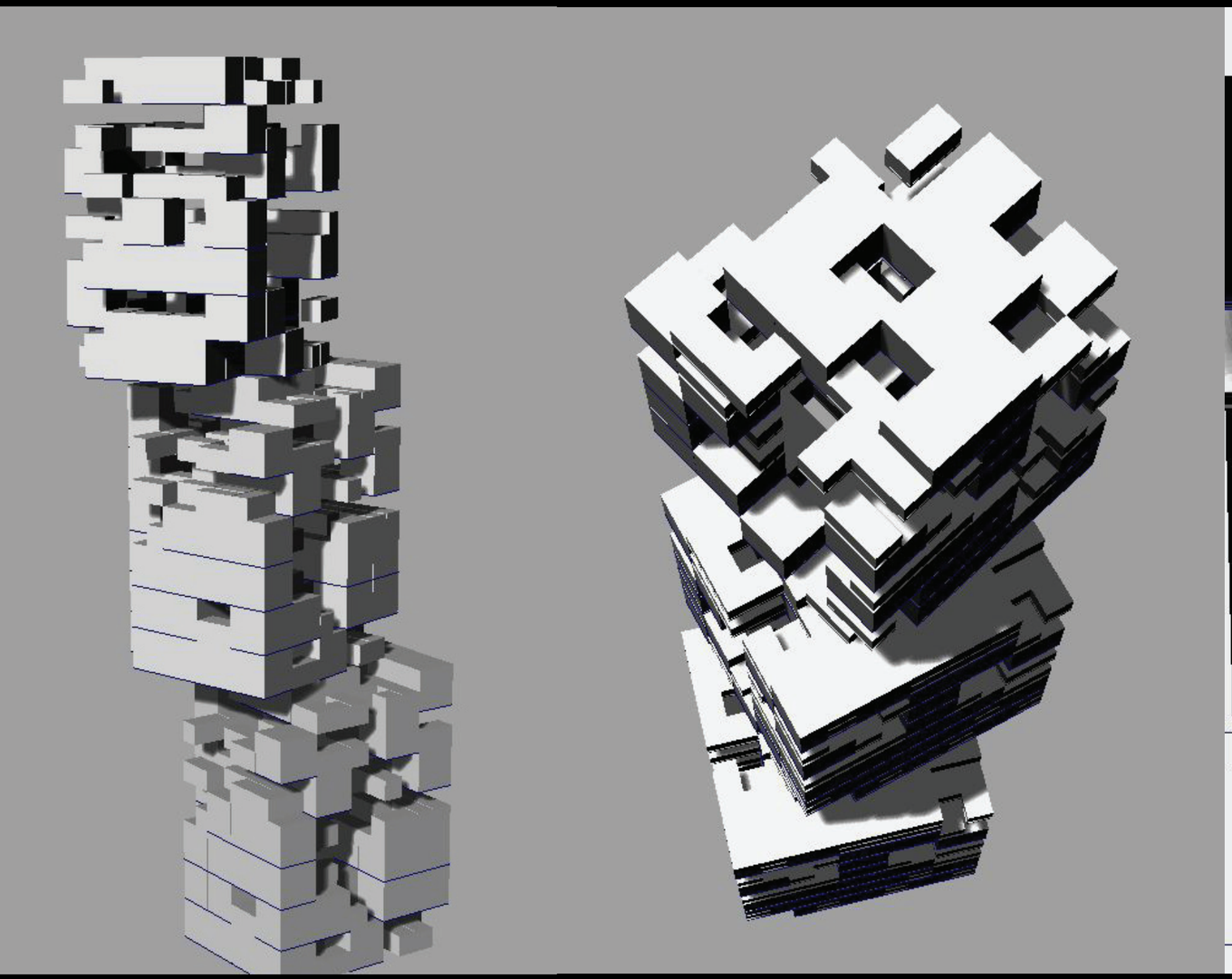


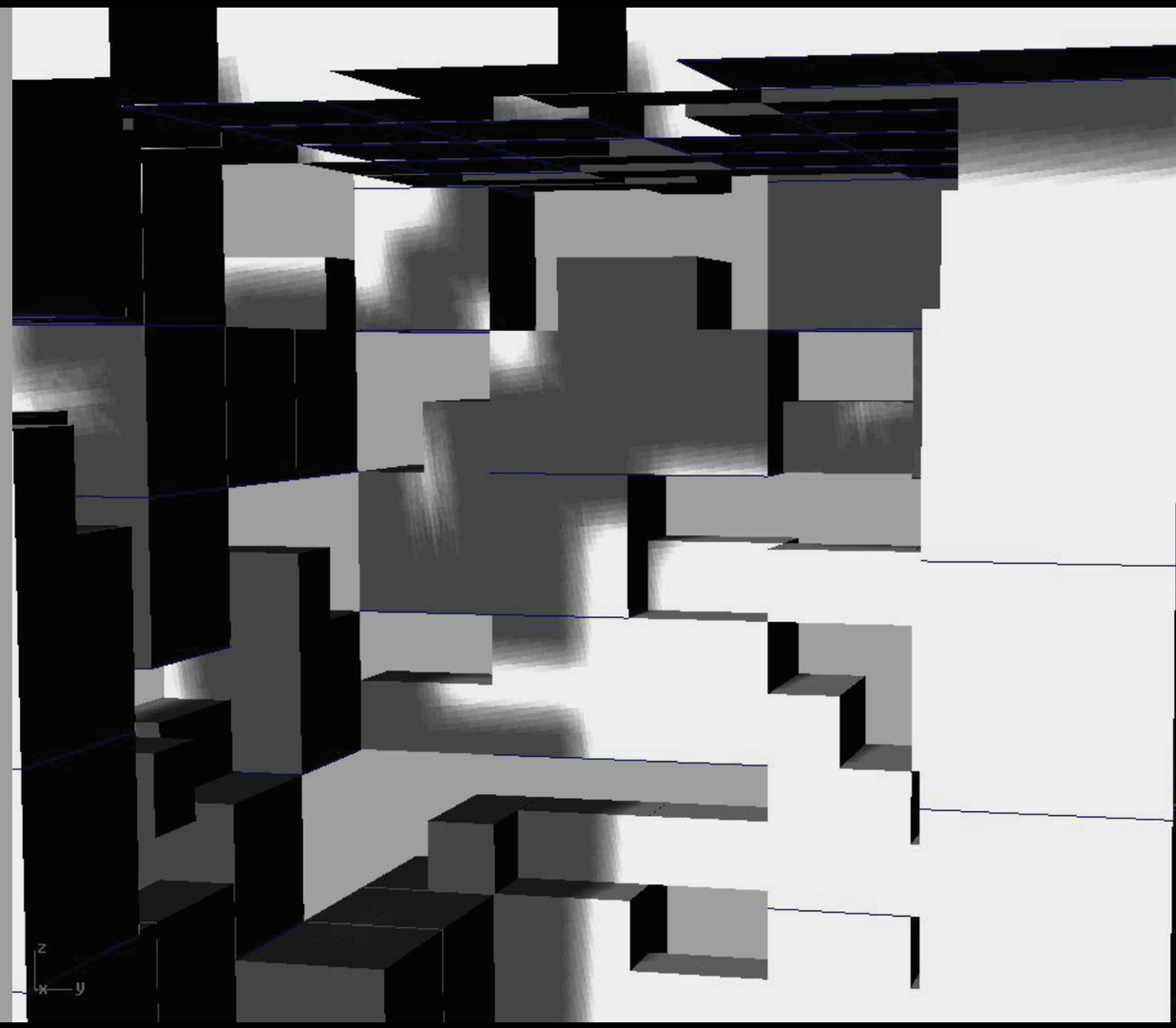
1
CONCEPT



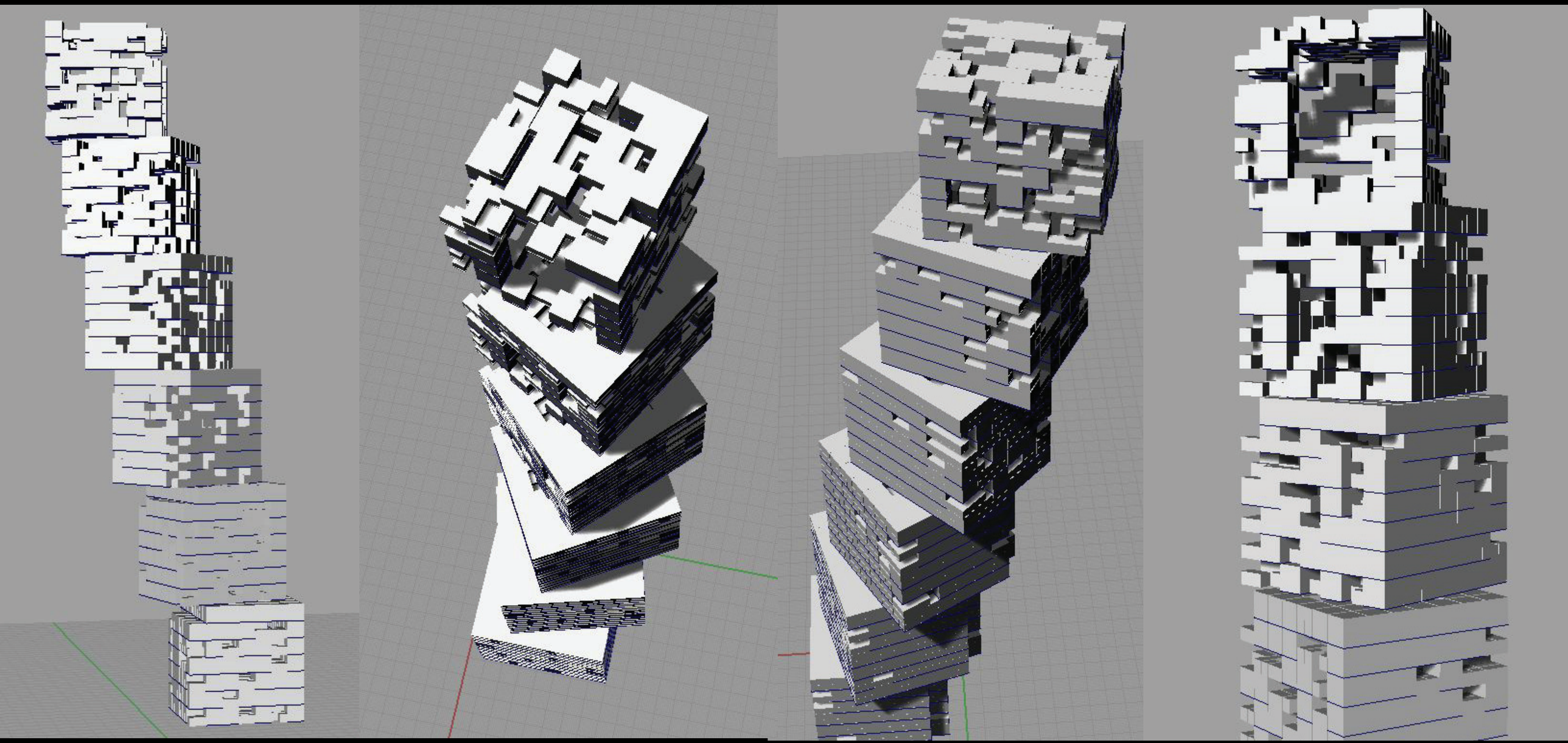


2
PROCESS SKETCHES



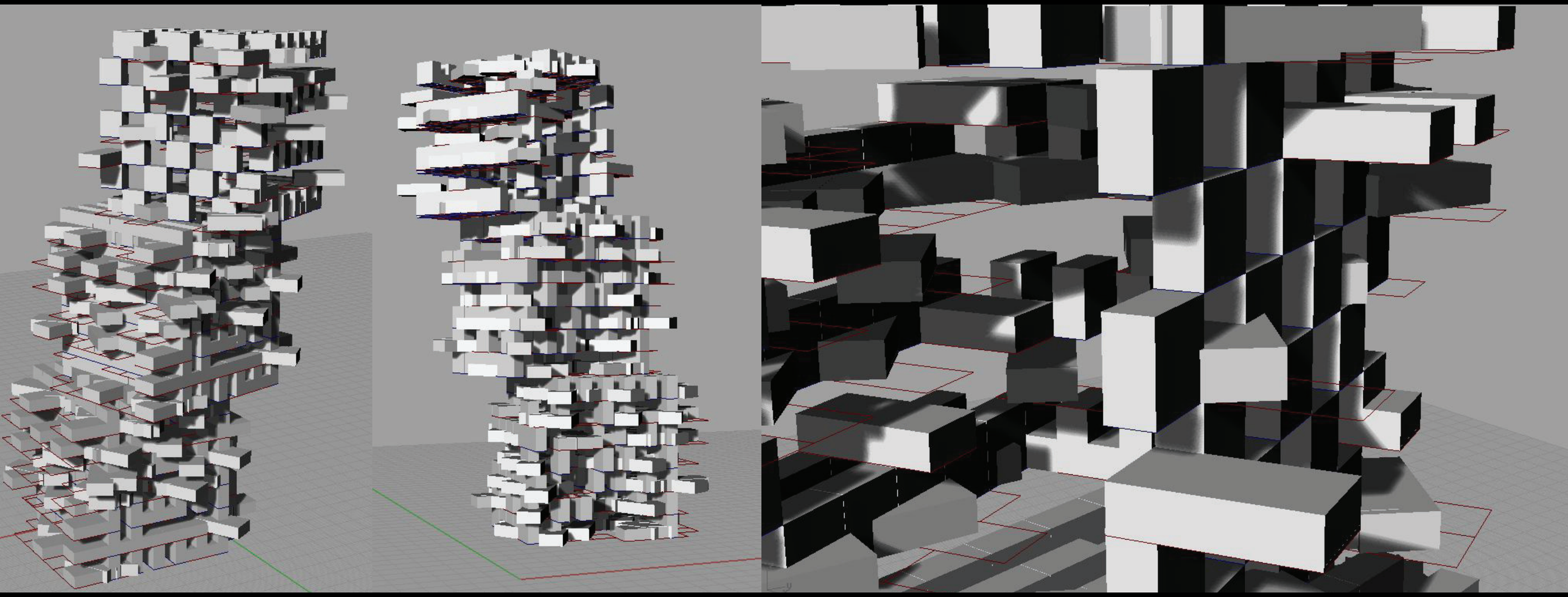


CA [STACK, ROTATE, 90]*

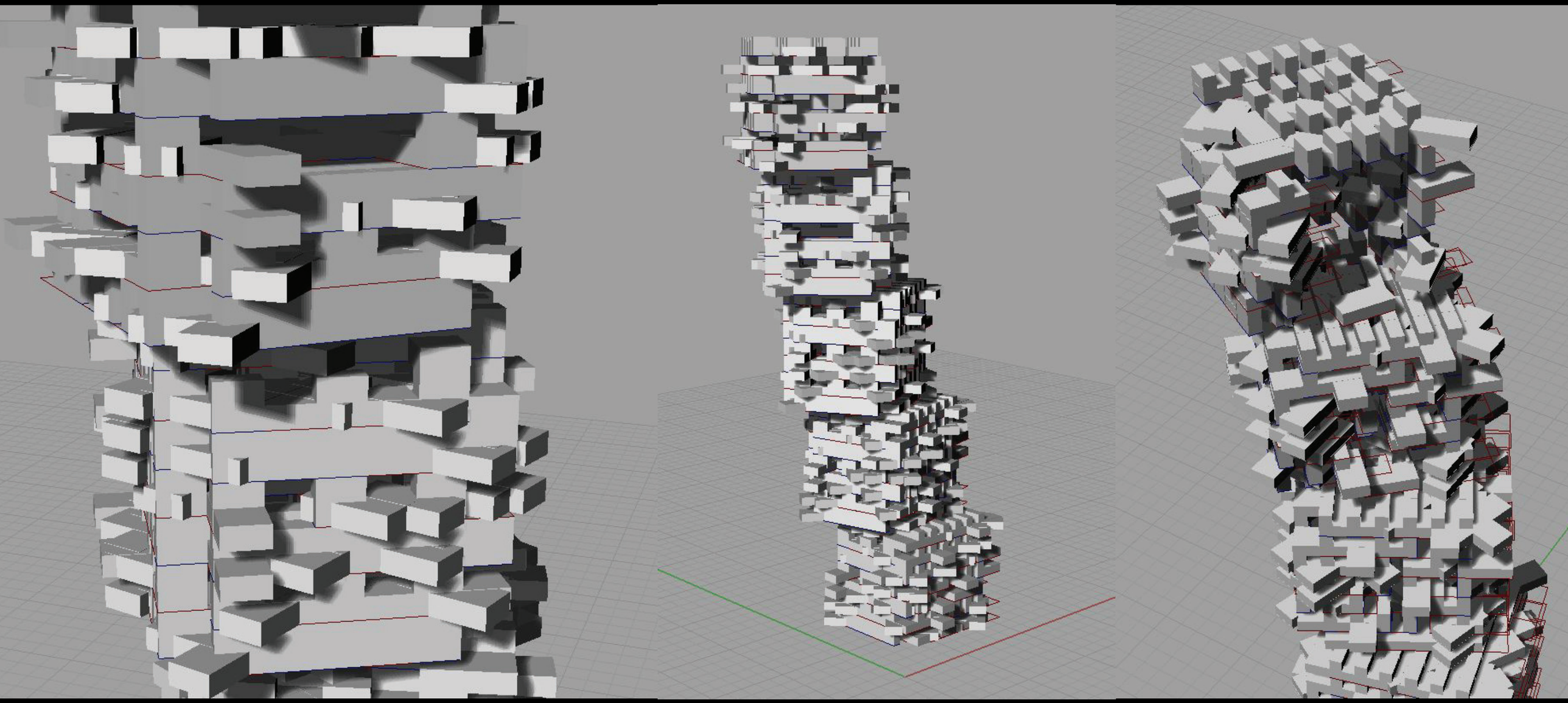


CA [STACK, ROTATE, 90]*

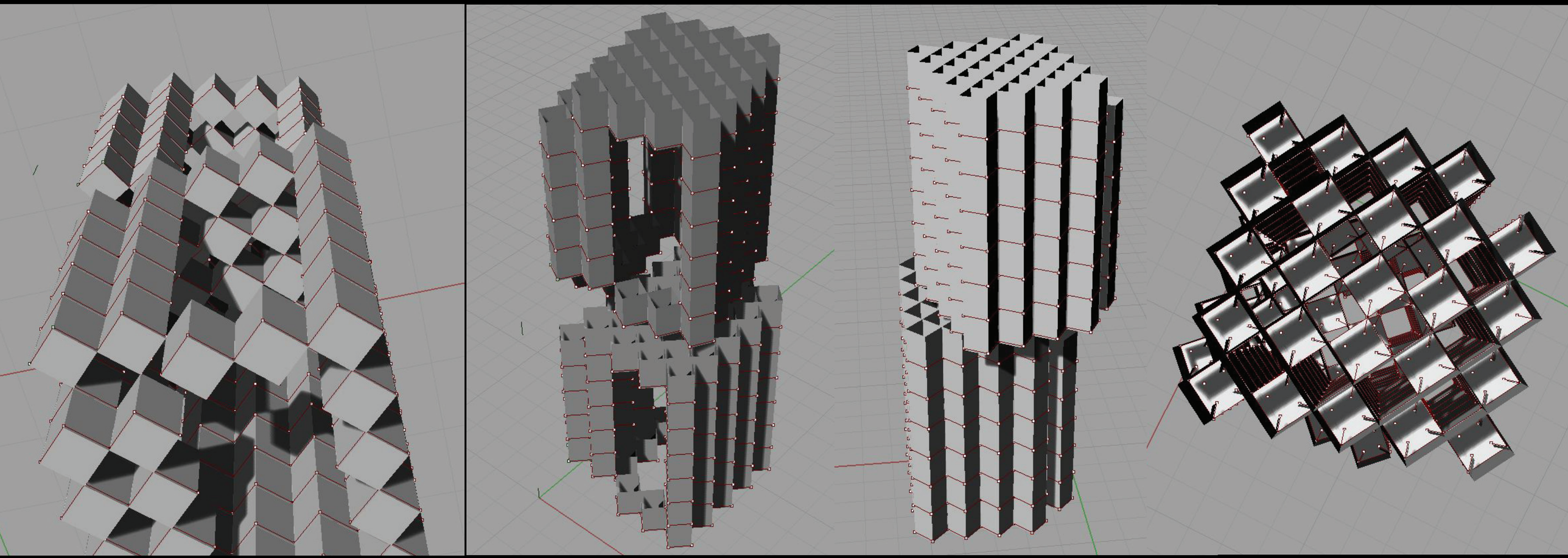
TESTING



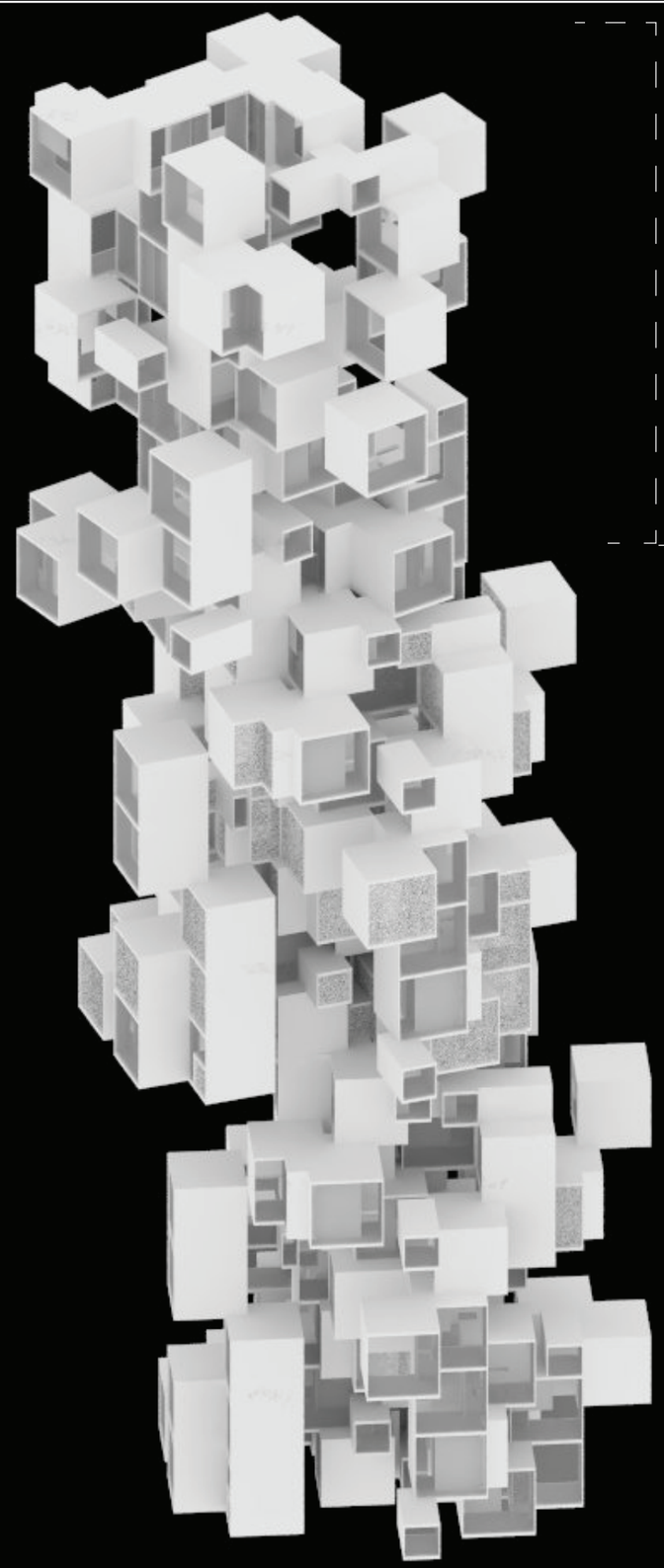
CA [STACK, ROTATE, 45 + 90*]*



CA [STACK, ROTATE, 60 + 90*]*



CA [STACK, ROTATE, 45]*

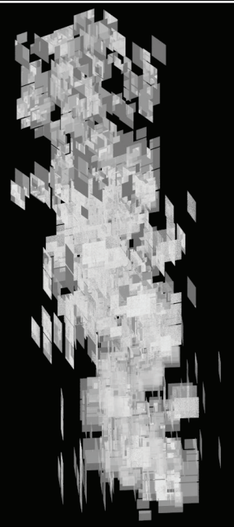


3

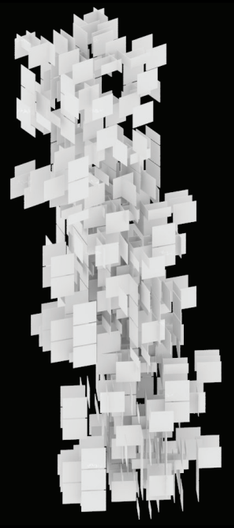
2

1

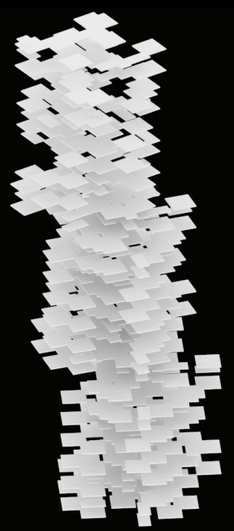
4
RENDERINGS



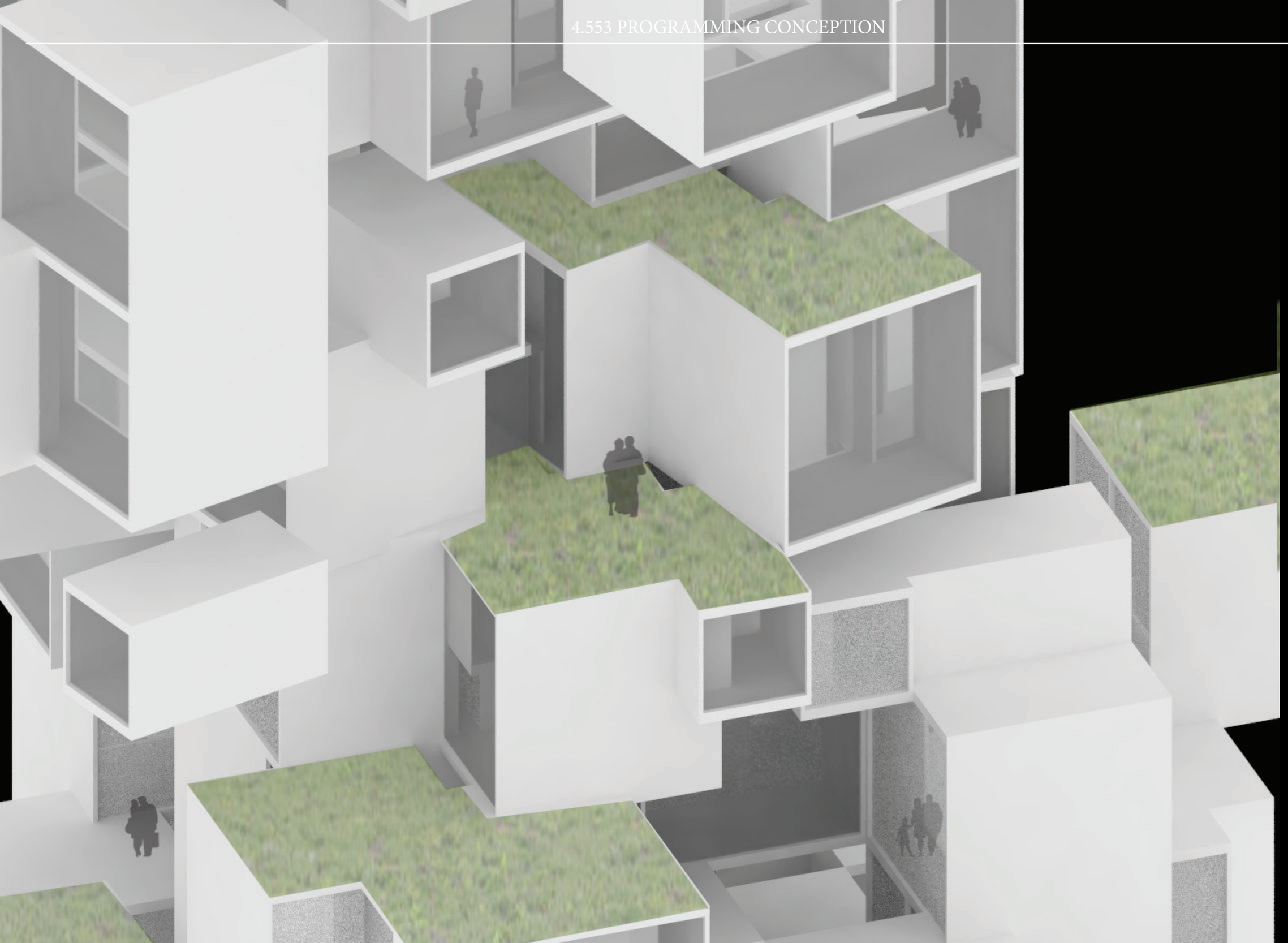
SKIN

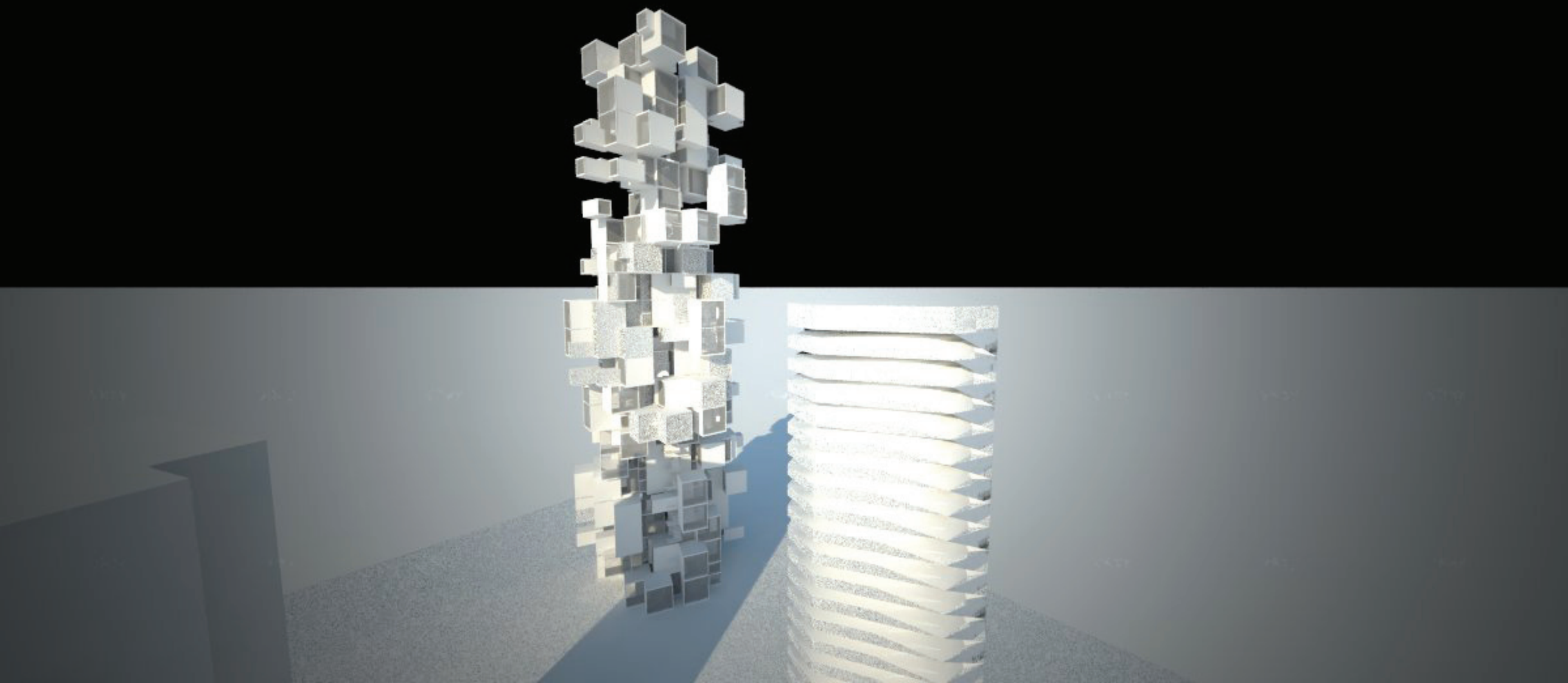


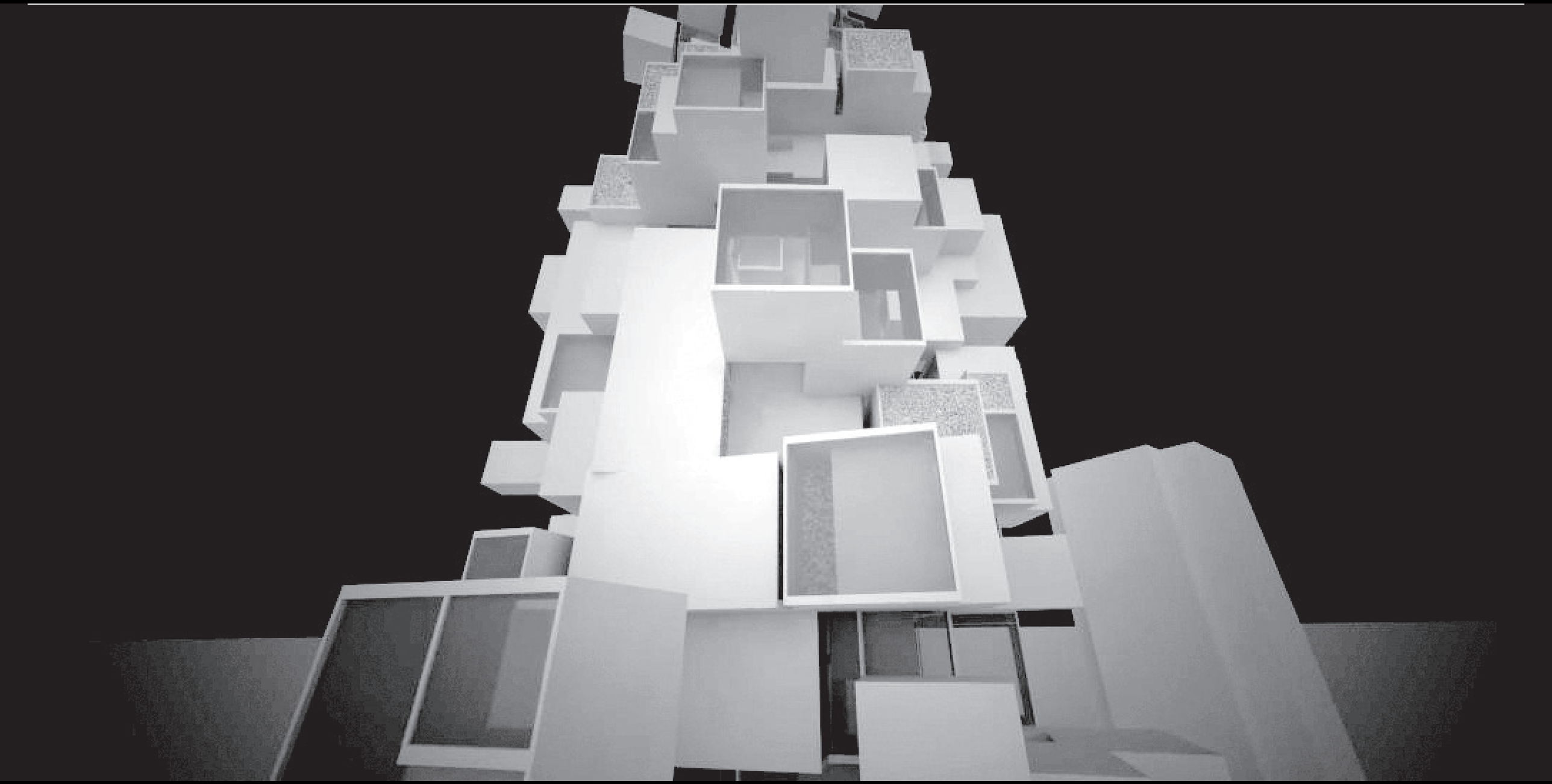
WALL



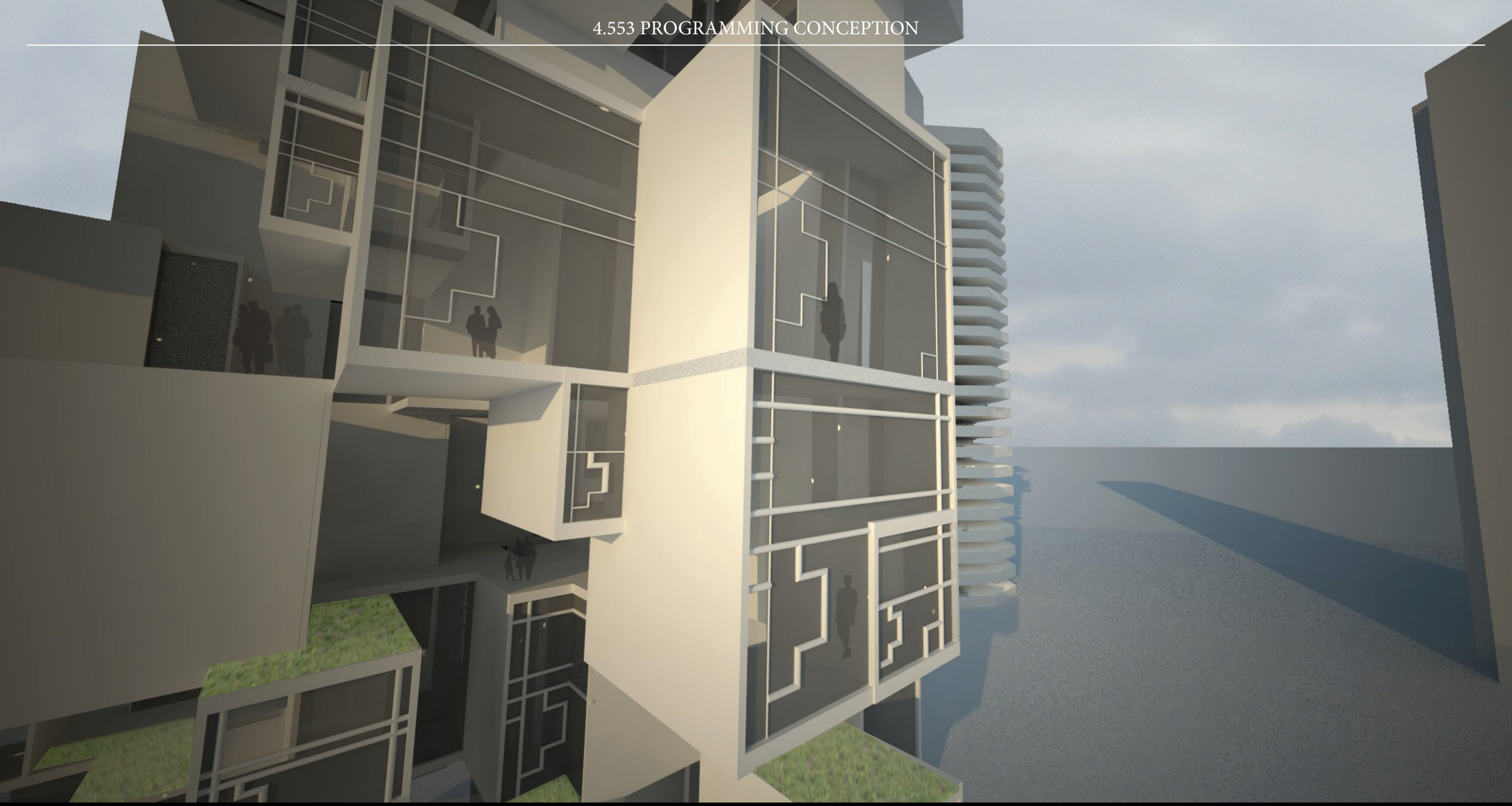
FLOOR







VIEW UPWARDS



VIEW WESTWARDS

HELLO BOSTON

```
import rhinoscriptsyntax as rs
from random import choice
import random
import SurfDivision as sd
reload (sd)

X= 5
Y= 5
Z= 8

scale_1 = 0.3

# 01. CELLS (CONDITION AND RULES)
class Cell():
    def __init__ (self,x,y,z):
        self.X = x
        self.Y = y
        self.Z = z
        self.Pos = [x,y,z]
        self.Visibility = False #True = cell's visible, False = not visible
        self.N = 0              #Number of neighbouring cells

    def getVisibility (self):
        return self.Visibility
    def setVisibility (self, v):
        self.Visibility = v

    def getN (self):
        return self.N
    def setN (self,n):
        self.N = n

    def getX (self):
        return self.X
    def getY (self):
        return self.Y
    def getZ (self):
        return self.Z

    def getPos (self):
        return self.Pos

    def updateVisibility (self):
        if (self.Visibility == True):   #when the cell is alive
            self.rule1_Visibility ()
        else: #(self.Visibility == False)  #when the cell is dead
            self.rule2_Population ()

    def rule1_Visibility(self):
        if (self.N < 7):
            self.setVisibility (False)
        elif (self.N < 15):
            self.setVisibility (True)
        else:
            self.setVisibility (False)

    def rule2_Population(self):
        if (self.N == 0 or self.N == 5 or self.N == 10):
            self.setVisibility (True)
        elif (self.N < 5):
            self.setVisibility (False)
        elif (self.N < 12):
            self.setVisibility (True)
        else:
            pass

# 02. SET OF CELLS
class SetOfCells ():
    def __init__ (self):
        self.ListCells = []             #List containing all cell classes
        self.generateCells ()           #these two functions automatically run
        self.setSeeds ()                #when the class is initiated

    def generateCells (self):           #this makes X x Y x Z cells
        xCellList = []                  #the outer list of a nested list (ListCells)
        for x in range (X):
            yCellList = []              #the inner list of a nested list (ListCells)
            for y in range (Y):
                zCellList = []
                for z in range (Z):
                    c = Cell (x,y,z)
                    zCellList.append (c)
                yCellList.append (zCellList)
            xCellList.append (yCellList)
        self.ListCells = xCellList      #returns the nested list into self.
ListCells

    def setSeeds (self):                #this function defines initial states of
cells
        for x in range (X):
            for y in range (Y):
                for z in range (Z):
                    cell = self.ListCells [x][y][z]
                    if (x > 1 and x < X and y > 0 and y < Y and z > 0 and z < Z):
#initial condition
                        cell.setVisibility (True)   #alive
                    else:
                        cell.setVisibility (False)  #dead

    def getCells (self):                #a getter for self.ListCells
        return self.ListCells

# 03. AUTOMATA CONTROLLER (MODEL)
class AutomataController ():
    def __init__ (self):
        self.ListCells = []
        self.SC = SetOfCells () #this will generate cells and contain them in
a list
        self.M = AutomataModel ()   #initiating AutomataModel

    def initiateModel (self):
        self.ListCells = self.SC.getCells ()
        self.M.setListCells (self.ListCells)
        self.M.repeatProcesses ()

# 04. AUTOMATA MODEL (VIEW)
class AutomataModel ():
    def __init__ (self):
        self.ListCells = []
        self.V = AutomataView ()

    def setListCells (self,lc):
        self.ListCells = lc

    def repeatProcesses (self):
        iterationNumber = 2 #<<<<<<<<<<<<Number of iteration process
of the CA>>>>>>>>>>>>>>>
        for i in range (iterationNumber):
            self.checkNeighboringCells ()
            self.updateCellVisibility ()
            self.V.setListCells (self.ListCells)
            points = self.V.getCoordinates ()
            centre = self.V.getCentre ()
            self.V.makeGeometries (points, centre)
            self.V.moveGeometries ()

    def checkNeighboringCells (self):
        for i in range (1,X-1):
            for j in range (1,Y-1):
                for k in range (1,Z-1):
                    cell = self.ListCells [i][j][k]
                    x = cell.getX ()
                    y = cell.getY ()
                    z = cell.getZ ()
                    n = self.calculateNumberOfNeighboringCells (x,y,z)
                    cell.setN (n)

    def calculateNumberOfNeighboringCells (self,x,y,z):
        cell0 = self.ListCells[x-1][y-1][z-1]
        cell1 = self.ListCells[x][y-1][z-1]
        cell2 = self.ListCells[x+1][y-1][z-1]
        cell3 = self.ListCells[x-1][y][z-1]
        cell4 = self.ListCells[x][y][z-1]
        cell5 = self.ListCells[x+1][y][z-1]
        cell6 = self.ListCells[x-1][y+1][z-1]
        cell7 = self.ListCells[x][y+1][z-1]
        cell8 = self.ListCells[x+1][y+1][z-1]

        cell9 = self.ListCells[x-1][y-1][z]
        cell10 = self.ListCells[x][y-1][z]
        cell11 = self.ListCells[x+1][y-1][z]
        cell12 = self.ListCells[x-1][y][z]
        cell13 = self.ListCells[x+1][y][z]
        cell14 = self.ListCells[x-1][y+1][z]
        cell15 = self.ListCells[x][y+1][z]
        cell16 = self.ListCells[x+1][y+1][z]

        cell17 = self.ListCells[x-1][y-1][z+1]
        cell18 = self.ListCells[x][y-1][z+1]
        cell19 = self.ListCells[x+1][y-1][z+1]
        cell20 = self.ListCells[x-1][y][z+1]
        cell21 = self.ListCells[x][y][z+1]
        cell22 = self.ListCells[x+1][y][z+1]
        cell23 = self.ListCells[x-1][y+1][z+1]
        cell24 = self.ListCells[x][y+1][z+1]
        cell25 = self.ListCells[x+1][y+1][z+1]

        v0 = cell0.getVisibility ()
        v1 = cell1.getVisibility ()
        v2 = cell2.getVisibility ()
        v3 = cell3.getVisibility ()
        v4 = cell4.getVisibility ()
        v5 = cell5.getVisibility ()
        v6 = cell6.getVisibility ()
        v7 = cell7.getVisibility ()
        v8 = cell8.getVisibility ()

        v9 = cell9.getVisibility ()
        v10 = cell10.getVisibility ()
        v11 = cell11.getVisibility ()
        v12 = cell12.getVisibility ()
        v13 = cell13.getVisibility ()
        v14 = cell14.getVisibility ()
        v15 = cell15.getVisibility ()
        v16 = cell16.getVisibility ()

        v17 = cell17.getVisibility ()
        v18 = cell18.getVisibility ()
        v19 = cell19.getVisibility ()
        v20 = cell20.getVisibility ()
        v21 = cell21.getVisibility ()
        v22 = cell22.getVisibility ()
        v23 = cell23.getVisibility ()
        v24 = cell24.getVisibility ()
        v25 = cell25.getVisibility ()

        sum = v0 + v1 + v2 + v3 + v4 + v5 + v6 + v7 + v8 + v9 + v10 + v11
+ v12 + v13 + v14 + v15 + v16 + v17 + v18 + v19 + v20 + v21 + v22 +
v23 + v24 + v25

        return sum

    def updateCellVisibility (self):
        for x in range (1,X-1):
            for y in range (1,Y-1):
                for z in range (1,Z-1):
                    cell = self.ListCells [x][y][z]
                    cell.updateVisibility ()

# 05. AUTOMATA VIEW
class AutomataView ():
    def __init__ (self):
        self.ListCells = []
        self.A = blockTypeA ()
        self.B = blockTypeB ()

    def setListCells (self,lc):
        self.ListCells = lc

    def getCoordinates (self):
        ptlist = []
        for x in range (X):
            for y in range (Y):
                for z in range (Z):
                    cell = self.ListCells [x][y][z]
                    visibility = cell.getVisibility ()
                    if visibility == True:
                        pos = cell.getPos()
                        ptlist.append (pos)
        length = len (ptlist)
        return ptlist

    def getCentre (self):
        ptlist = []
        for x in range (X):
            for y in range (Y):
                for z in range (Z):
                    cell = self.ListCells [x][y][z]
                    visibility = cell.getVisibility ()
                    if visibility == True:
                        pos = cell.getPos ()
                        shift = 0.5
```

5
CODES

```
            centre = [pos[0]+shift, pos[1]-shift, pos[2]] #pos[1]-shift creates cantilevering options
            ptlist.append (centre)
        return ptlist

    def makeGeometries (self, points, centre):
        length = len (points)
        percent = scale_1 * length # scaling ration
        minority = int(percent) # count of input percentage of all POINTS
        majority = length - minority # count of remaining percentage of all POINTS

#for Block Type A Units
        typeA_list = []
        for i in range(minority):
            randomvalues = random.randint (0, length-1)
            typeA_list.append (randomvalues)

        typeA_list = list (set(typeA_list)) #eliminating duplicates
        typeA_list.sort() #re-list in ascending order
        print typeA_list

#for Block Type B Units
        typeB_list = []
        for i in range (length):
            typeB_list.append (i)
        print typeB_list

        for x in typeA_list:
            typeB_list.remove (x)
        self.A.generateUnitA (typeA_list, points, centre)
        self.B.generateUnitB (typeB_list, points, centre)

    def moveGeometries(self):
        rs.Sleep(1000)
        allObjIDs = rs.AllObjects ()
        centre_point = [0,0,0]
        rotatedObjsIDs = rs.RotateObjects (allObjIDs, centre_point, 15, None, True ) #rotate all CA block(s) by certain angle
        trans2 = [0,0,7]
        movedObjIDs = rs.MoveObjects (rotatedObjsIDs, trans2) #move CA block(s) upwards

#07. GENERATING TYPE A UNITS
class blockTypeA():
    def __init__(self):
        self.slab = makeFloors()
        self.wall = wall()
        self.wall_2 = wall_2()
        self.randomVariant = randomVariant()

        self.slabthickness = 0.05
        self.wallthickness = 0.02
        self.glassoffset = 0.02

        self.floorheight = 0.5
        self.X_length_A = 0.5
        self.Y_length_A = 1

    def generateUnitA (self, unitnumber, points, centre):

        for r in unitnumber :
            rs.AddLayer("Block Type A", [100,0,0])
            rs.CurrentLayer("Block Type A")

            x = points [r][0]
            y = points [r][1]
            z = points [r][2]

            pt0 = [x, y, z]
            pt1 = [x, y + self.Y_length_A, z]
            pt2 = [x + self.X_length_A, y + self.Y_length_A, z]
            pt3 = [x + self.X_length_A, y, z]

            rectangle = rs.AddRectangle(points[r], self.X_length_A, self.Y_length_A)

            slabs = self.slab.makeFloors (rectangle, self.slabthickness, self.floorheight)
            wall1 = self.wall.wall(pt0, self.wallthickness, self.Y_length_A, self.floorheight, self.slabthickness)
            wall2 = self.wall.wall(pt2, -self.wallthickness, -self.Y_length_A, self.floorheight, self.slabthickness)

            glass1 = self.wall_2.glass(pt2, pt1, pt0, self.glassoffset, self.floorheight, self.slabthickness)
            glass2 = self.wall_2.glass(pt0, pt3, pt1, self.glassoffset, self.floorheight, self.slabthickness)

            angle = self.randomVariant.randomGenerator()

            verticalshift = [0, 0.5]
            shift = choice(verticalshift)
            path = [0, 0, shift]

            rs.RotateObjects(slabs, centre[r], angle)
            rs.RotateObjects(wall1, centre[r], angle)
            rs.RotateObjects(wall2, centre[r], angle)
            rs.RotateObjects(glass1, centre[r], angle)
            rs.RotateObjects(glass2, centre[r], angle)

            rs.MoveObjects(slabs, path)
            rs.MoveObjects(wall1, path)
            rs.MoveObjects(wall2, path)
            rs.MoveObjects(glass1, path)
            rs.MoveObjects(glass2, path)

            rs.DeleteObject(rectangle)

#08. GENERATING TYPE B UNITS
class blockTypeB():
    def __init__(self):
        self.slab = makeFloors()
        self.wall = wall()
        self.wall_2 = wall_2()
        self.randomVariant = randomVariant()

        self.slabthickness = 0.05
        self.wallthickness = 0.02
        self.glassoffset = 0.02

        self.floorheight = 1
        self.X_length_B = 1
        self.Y_length_B = 1

    def generateUnitB (self, unitnumber, points, centre):
        for r in unitnumber :
            rs.AddLayer("Block Type B", [0,100,0])
            rs.CurrentLayer("Block Type B")

            x = points [r][0]
            y = points [r][1]
            z = points [r][2]

            pt0 = [x, y, z]
            pt1 = [x, y + self.Y_length_B , z]
            pt2 = [x + self.X_length_B, y + self.Y_length_B, z]
            pt3 = [x + self.X_length_B, y, z]

            rectangle = rs.AddRectangle(points[r], self.X_length_B, self.Y_length_B)
            slab = self.slab.makeFloors (rectangle, self.slabthickness, self.floorheight)

            wall1 = self.wall.wall(pt0, self.wallthickness, self.Y_length_B, self.floorheight, self.slabthickness)
            wall2 = self.wall.wall(pt2, -self.wallthickness, -self.Y_length_B, self.floorheight, self.slabthickness)

            glass1 = self.wall_2.glass(pt2, pt1, pt0, self.glassoffset, self.floorheight, self.slabthickness)
            glass2 = self.wall_2.glass(pt0, pt3, pt1, self.glassoffset, self.floorheight, self.slabthickness)

            angle = self.randomVariant.randomGenerator()

            slab = rs.RotateObjects(slab, centre[r], angle)
            wall1 = rs.RotateObjects(wall1, centre[r], angle)
            wall2 = rs.RotateObjects(wall2, centre[r], angle)
            glass1 = rs.RotateObjects(glass1, centre[r], angle)
            glass2 = rs.RotateObjects(glass2, centre[r], angle)

            rs.DeleteObject(rectangle)

#09. CREATING FLOORS
class makeFloors():
    def __init__(self):
        pass
    def makeFloors(self, curve, slabthickness, floorheight):
        path1 = rs.AddLine([0,0,0], [0,0,-slabthickness]) # Floor Thickness
        extrusion = rs.ExtrudeCurve (curve, path1)
        rs.CapPlanarHoles(extrusion)
        copy = rs.CopyObject(extrusion, [0,0,floorheight])

        return extrusion, copy
        rs.DeleteObject(curve)
        rs.DeleteObject(path1)

#10. CREATING WALLS TYPE ONE
class wall():
    def __init__(self):
        pass
    def wall(self, point1, wallthickness, walllength, floorheight, slabthickness):
        rs.AddLayer("Wall", [0,0,0])
        rs.CurrentLayer("Wall")

        rectangle = rs.AddRectangle(point1, wallthickness, walllength)
        path = rs.AddLine([0,0,0], [0,0,floorheight-slabthickness])

        extrusion = rs.ExtrudeCurve (rectangle, path)
        rs.CapPlanarHoles(extrusion)
        return extrusion
        rs.DeleteObjects(rectangle)
        rs.DeleteObject(path)

#11. SURFACE DIVISION
class Make3DSystem ():
    def __init__ (self, surfID):
        self.pt = [0,0,0]
        self.dir = [1,0,0]
        self.fList = []
        uDiv = 10
        vDiv = 10
        s = sd.SurfDivision (uDiv,vDiv, surfID)
        self.ptList = s.getList ()

    def A (self):
        ptS = self.pt
        dir = self.dir
        ptE = rs.VectorAdd (ptS,dir)
        #id = rs.AddLine (ptS, ptE)
        u0 = int (ptS[0])
        v0 = int (ptS[1])
        u1 = int (ptE[0])
        v1 = int (ptE[1])

        pt0 = self.ptList[u0][v0]
        pt1 = self.ptList[u1][v1]
        ptsTemp = [pt0,pt1]
        rs.AddLine(pt0, pt1)
        rs.AddCylinder (pt0, pt1, 0.015)
        self.pt = ptE

    def B (self):
        ptS = self.pt
        dir = self.dir
        ptE = rs.VectorAdd (ptS,dir)
        #id = rs.AddLine (ptS, ptE)
        u0 = int (ptS[0])
        v0 = int (ptS[1])
        u1 = int (ptE[0])
        v1 = int (ptE[1])

        pt0 = self.ptList[u0][v0]
        pt1 = self.ptList[u1][v1]
        ptsTemp = [pt0,pt1]
        rs.AddCylinder (pt0, pt1, 0.015)
        rs.AddLine(pt0, pt1)
        self.pt = ptE

    def Plus (self):
        d = self.dir
        axis = [0,0,1]
        newD = rs.VectorRotate(d, 90, axis)
        self.dir = newD

    def Minus (self):
        d = self.dir
        axis = [0,0,1]
        newD = rs.VectorRotate(d, -90, axis)
```

CODES

```
    self.dir = newD

def applyRule (self):
    newList = []
    index = self.fList          #to get the existing list for functions

    for f in index:             #for every function
        fName = f.__name__      #get the existing name in the list

        AName = self.A.__name__ #get the function name (A)
        BName = self.B.__name__

        if fName == AName:      #if the name is A, then make a rule
            newList.append (self.A)
            newList.append (self.Minus)
            newList.append (self.B)
            newList.append (self.Minus)
            newList.append (self.A)
            newList.append (self.Plus)
            newList.append (self.B)
            newList.append (self.Minus)
            newList.append (self.B)

        elif fName == BName:
            newList.append (self.B)
            newList.append (self.Plus)
            newList.append (self.A)
            newList.append (self.Minus)
            newList.append (self.B)
            newList.append (self.Plus)
            newList.append (self.A)

        else:                   #if it is Plus or Minus
            newList.append (f)  #just add it in the list

        self.fList = newList    #update the function list


    def makeRecursiveRule (self,index):     #this will call the self.Rule()
recursively
        if index == 0:
            pass                        #if the index = 0, do nothing
        else:
            newIndex = index -1             #if the index >1, run recursion
            self.applyRule()
            self.makeRecursiveRule (newIndex)
    def runList (self):
        for f in self.fList:
            f()

#Add a single F in a list
    def addFInList (self):
        self.fList = [self.A,self.B]

#Run to create a 3D L-System!
    def run3DSystem (self, index):
        if index <0:                    #if the index <0, do nothing
            pass
        elif index == 0:                  #when the index is 0
            self.addFInList ()              #add one F in the list
        elif index > 0:                   #if index >0
            self.addFInList ()              #add one F in the list
            self.makeRecursiveRule (index)  #recursively generate rules
            self.runList ()                 #run all the functions in the list

#12. CREATING WALLS TYPE TWO
class wall_2():
    def __init__(self):
        pass

    def glass (self, point1, point2, directionpoint, offset, floorheight,
slabthickness):
        rs.AddLayer("Glass", [25,25,25])
        line = rs.AddLine(point1, point2)
        offset = rs.OffsetCurve(line, directionpoint, offset)

        path = rs.AddLine([0,0,0], [0,0,floorheight-slabthickness])

        extrusion = rs.ExtrudeCurve (offset, path)

        rs.ObjectLayer(extrusion, "Glass")
        return extrusion

#13. VARYING BLOCK ORIENTATION
class randomVariant():
    def __init__(self):
        pass

    def randomGenerator(self):
        rotation = [45,135,225,315]
        angles = choice(rotation)
        return angles

#14. MAKING WINDOW PATTERN
class WindowPattern():
    def __init__(self):
        self.ListOfWindows = None

    def setList(self):
        glassLayer = "Glass"
        self.ListOfWindows = rs.ObjectsByLayer(glassLayer)

    def makeWindowPattern(self):
        self.setList()
        size = len(self.ListOfWindows)
        index = range(size)
        for i in index:

            surfID = self.ListOfWindows[i]
            st = Make3DSystem(surfID)
            st.run3DSystem (2)      #insert number as input for generating a
3D System

#15. RUN AUTOMATA CONTROLLER
rs.EnableRedraw (False)
print "ROTATED CA TOWER!"
ac = AutomataController()
ac.initiateModel()

w = WindowPattern()
w.makeWindowPattern()
```

CODES