Menger 3D Algorithm

1 set two points that define the bounding box of a cube,

2 divide the cube into 27 sub-cubes,

3 discard the data for the center sub-cube,

4 store the bounding boxes of the remaining 26 sub-cubes.

By following steps 1 to 4 you can generate the first sub-division level for the 3D Sponge.

The next step is to repeat instrustions 2 to 4 until the recursion depth is zero, and this

will generate the remaining sub-division levels.

Below is the python code for creating a Menger 3D Sponge RIB archive node.

In addition I used the ri\_utils.py file that I got from the class website

in order to make the Menger code work properly. The ri\_utils.py code can be

found with the menger code in the text box below.

RI\_Utils.py code (Read first, then read Menger code)

# ri\_utils.py

# A collection of procs that generate strings containing the

# RenderMan description of a surface. For example,

#

# import ri\_utils

# verts = [ [-1,0,1], [1,0,1], [1,0,-1] ]

# print ri\_utils.Polygon(verts)

#

# will generate the following Rib statement:

# Polygon "P" [-1.0 0.0 1.0 1.0 0.0 1.0 1.0 0.0 -1.0]

#

# Malcolm Kesson Jan 20 2013

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def Cube(bbox, index):

 pnts = []

 minX,minY,minZ,maxX,maxY,maxZ = bbox

 rib = 'Attribute "identifier" "float id" [' + str(index) + ']\n'

 rib += 'PointsGeneralPolygons [1 1 1 1 1 1] '

 rib += '[4 4 4 4 4 4]\n'

 rib += '\t\t[0 1 3 2 2 3 5 4 4 5 7 6 6 7 1 0 1 7 5 3 6 0 2 4]\n'

 rib += '\t\t"P" ['

 pnts.append('%1.3f %1.3f %1.3f' % (minX,minY,maxZ))

 pnts.append(' %1.3f %1.3f %1.3f' % (maxX,minY,maxZ))

 pnts.append(' %1.3f %1.3f %1.3f' % (minX,maxY,maxZ))

 pnts.append(' %1.3f %1.3f %1.3f' % (maxX,maxY,maxZ))

 pnts.append(' %1.3f %1.3f %1.3f' % (minX,maxY,minZ))

 pnts.append(' %1.3f %1.3f %1.3f' % (maxX,maxY,minZ))

 pnts.append(' %1.3f %1.3f %1.3f' % (minX,minY,minZ))

 pnts.append(' %1.3f %1.3f %1.3f' % (maxX,minY,minZ))

 rib += ''.join(pnts)

 rib += ']\n'

 return rib

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def \_\_cube\_edges(bbox):

 x0,y0,z0,x1,y1,z1 = bbox

 edges = []

 # lower edges

 edges.append([ [x0,y0,z0], [x0,y0,z1] ]) # edge 0\_1

 edges.append([ [x0,y0,z1], [x1,y0,z1] ]) # edge 1\_2

 edges.append([ [x1,y0,z1], [x1,y0,z0] ]) # edge 2\_3

 edges.append([ [x1,y0,z0], [x0,y0,z0] ]) # edge 3\_0

 # upper edges

 edges.append([ [x0,y1,z0], [x0,y1,z1] ]) # edge 4\_5

 edges.append([ [x0,y1,z1], [x1,y1,z1] ]) # edge 5\_6

 edges.append([ [x1,y1,z1], [x1,y1,z0] ]) # edge 6\_7

 edges.append([ [x1,y1,z0], [x0,y1,z0] ]) # edge 7\_4

 # vertical edges

 edges.append([ [x0,y0,z0], [x0,y1,z0] ]) # edge 0\_4

 edges.append([ [x0,y0,z1], [x0,y1,z1] ]) # edge 1\_5

 edges.append([ [x1,y0,z1], [x1,y1,z1] ]) # edge 2\_6

 edges.append([ [x1,y0,z0], [x1,y1,z0] ]) # edge 3\_7

 return edges

def CubeEdges(bbox, width):

 edges = \_\_cube\_edges(bbox)

 rib = ''

 for edge in edges:

 pnts = []

 begin,end = edge

 x,y,z = begin

 X,Y,Z = end

 rib += '\tCurves "linear" [2] "nonperiodic" \n'

 rib += '\t\t"P" ['

 pnts.append('%1.3f %1.3f %1.3f' % (x,y,z))

 pnts.append(' %1.3f %1.3f %1.3f' % (X,Y,Z))

 rib += ''.join(pnts)

 rib += '] "constantwidth" [%1.3f]\n' % width

 return rib

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def Polygon(verts, index):

 pnts = []

 rib = 'Attribute "identifier" "float id" [' + str(index) + ']\n'

 rib += 'Polygon "P" ['

 for vert in verts:

 x,y,z = vert

 pnts.append('%1.3f %1.3f %1.3f ' % (x,y,z))

 rib += ''.join(pnts)

 rib += ']\n'

 return rib

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def HierarchicalSubdivisionMesh(mesh, index):

 loops = [] # a list of the number of vertices of each face

 vertLUT = {} # a sequence of unique vertices

 for poly in mesh:

 loops.append(' %d' % len(poly))

 for vert in poly:

 vertLUT[vert] = vert

 # Convert the LUT to a list so that indexing can be used

 pnts = []

 for item in vertLUT.keys():

 pnts.append(item)

 indices = []

 for poly in mesh:

 for vert in poly:

 indices.append(' %d' % pnts.index(vert))

 rib = 'Attribute "identifier" "float id" [' + str(index) + ']\n'

 rib += 'HierarchicalSubdivisionMesh "catmull-clark" \n['

 rib += ''.join(loops)

 rib += ']\n['

 rib += ''.join(indices)

 rib += ']\n'

 rib += '["creasemethod" "facevaryingpropagatecorners" "interpolateboundary"] [0 0 1 1 0 0 1 0 0] [1 1] [] ["chaikin"]\n'

 rib += '"P" [\n'

 pntstr = []

 for pnt in pnts:

 x,y,z = pnt

 pntstr.append('%1.4f %1.4f %1.4f ' % (x,y,z))

 rib += ''.join(pntstr)

 rib += ']\n'

 return rib

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def PointsGeneralPolygons(mesh, index):

 faces = []

 loops = []

 vertLUT = {}

 for poly in mesh:

 faces.append(' 1')

 loops.append(' %d' % len(poly))

 for vert in poly:

 vertLUT[vert] = vert

 pnts = []

 for item in vertLUT.keys():

 pnts.append(item)

 pnts.sort()

 indices = []

 for poly in mesh:

 for vert in poly:

 indices.append(' %d' % pnts.index(vert))

 rib = 'Attribute "identifier" "float id" [' + str(index) + ']\n'

 rib += 'PointsGeneralPolygons ['

 rib += ''.join(faces)

 rib += ']\n['

 rib += ''.join(loops)

 rib += ']\n['

 rib += ''.join(indices)

 rib += '] \n"P" [\n'

 pntstr = []

 for pnt in pnts:

 x,y,z = pnt

 pntstr.append('%1.4f %1.4f %1.4f ' % (x,y,z))

 rib += ''.join(pntstr)

 rib += ']\n'

 return rib

#\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def Cylinder(bbox, index):

 x,y,z,X,Y,Z = bbox

 xrad = (X - x) / 2

 zrad = (Z - z) / 2

 rad = (xrad + zrad) / 2

 height = Y - y

 rib = 'TransformBegin\n'

 rib += '\tTranslate %1.3f %1.3f %1.3f \n' % (x,y,z)

 rib += '\tRotate 90 1 0 0\n'

 rib += '\tAttribute "identifier" "float id" [' + str(index) + ']\n'

 rib += '\tCylinder %1.3f 0 %1.3f 360\n' % (rad,height)

 rib += 'TransformEnd\n'

 return rib

Menger Code------------------------------------------------------------

import ri\_utils

from math import sqrt

class Menger3D:

 def \_\_init\_\_(self, bbox, depth, listOfHoles):

 self.deletedCubes = [] # list of deleted cubes

 self.retainedCubes = []

 listOfHoles.sort()

 listOfHoles.reverse()

 self.holeLUT = listOfHoles

 self.bbox = bbox # minx,miny,minz, maxx,maxy,maxz

 self.depth = depth

 self.divide(bbox, depth) # our recursive routine

 #\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

 # Given the minimum x,y,z and maximum x,y,z coordinates

 # of a bounding box this proc returns the bouding box

 # coordinates of a "row" of three cubes.

 def row(self, x0,y0,z0, w,h,d):

 x,y,z = x0,y0,z0

 X,Y,Z = x + w, y + h, z + d

 cubes = []

 for n in range(3):

 cube = [x,y,z, X,Y,Z]

 cubes.append(cube)

 z,Z = z + d, Z + d

 return cubes

 #\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

 # A recursive proc that subdivides a bounding box into

 # 27 sub-cubes. Each time the proc is called the arg

 # "depth" is decremented. Recursion terminates when its

 # value becomes zero.

 def divide(self, bbox, depth):

 if depth == 0:

 self.retainedCubes.append(bbox)

 return []

 x0,y0,z0,x1,y1,z1 = bbox

 w = float(x1 - x0)/3

 h = float(y1 - y0)/3

 d = float(z1 - z0)/3

 x,y,z = x0,y0,z0

 cubes = []

 for layer in range(3):

 x = x0

 for rows in range(3):

 cubes.extend(self.row(x,y,z,w,h,d))

 x = x + w

 y = y + h

 cubes = self.delete(cubes)

 # Recursion\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

 for cube in cubes:

 self.divide(cube, depth - 1)

 return cubes

 #\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

 # Uses the indices in the holeLUT to remove specific cubes

 # from the list of 27 cubes in the "cubes" arg.

 def delete(self,cubes):

 for n in range(len(self.holeLUT)):

 index = self.holeLUT[n]

 if index < len(cubes):

 deleted = cubes.pop(self.holeLUT[n])

 self.deletedCubes.append(deleted)

 return cubes

 #\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

 # Write a RenderMan archive rib file for the menger cubes

 # or for the cubes that were removed by the delete method.

 def writeAsCubes(self, rib\_path, cube\_type='retained'):

 f = open(rib\_path,'w')

 bboxStr = ' '.join(map(str, self.bbox))

 f.write('#bbox: %s\n' % bboxStr)

 if cube\_type == 'retained':

 cubes = self.retainedCubes

 else:

 cubes = self.deletedCubes

 counter = 1

 for cube in cubes:

 f.write(ri\_utils.Cube(cube, counter))

 counter += 1

 f.close()

 return len(cubes)

 #\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

 # Not used in this example implementation but could be

 # used to cull cubes on the basis of their distance from

 # a central location - to create a menger sphere.

 def distance(self, p1, p2):

 x = p1[0]-p2[0]

 y = p1[1]-p2[1]

 z = p1[2]-p2[2]

 return sqrt(x \* x + y \* y + z \* z)

#=======================================================

if \_\_name\_\_=="\_\_main\_\_":

 bounds = [-1,0,-1, 1,2,1]

 removals = [22,16,14,13,12,10,4]

 menger3d = Menger3D(bounds, 4, removals)

 menger3d.writeAsCubes('/RIB Path/menger3d.rib')

 menger3d.writeAsCubes('/RIB Path/menger3d\_holes.rib','')