Procedural Generation of Sculptural Forms

George W. Hart
Computer Science Department
Stony Brook University
Stony Brook, NY, USA
E-mail: `george@georgehart.com`

Abstract

Software methods in *Mathematica* are presented for creating visually interesting 3D forms to be produced on solid freeform fabrication (SFF) machines. Concise examples demonstrate fundamental techniques of procedural design that can be combined and extended in many creative ways. This overview should interest non-programmers who attend mathematical art exhibits of SFF objects and wonder where to begin to produce their own ideas.

Introduction

A *Special Topics* course I taught at Stony Brook University in Fall 2007 covered procedural generation of three-dimensional forms, with an emphasis on making triangulated manifold boundaries suitable for solid freeform fabrication (SFF). I previously taught similar courses using Java and Java3D, but *Mathematica* version 6 (abbreviated *M* here) [1] became available just before the class began, and I decided to use it as the programming environment. This paper presents core material other teachers and students can use. The main idea is that short programs can be written quickly to create geometric sculpture or visually interesting mathematical models. A good way to learn a new programming language is to study short examples and modify them. The examples provide a starting point of programs to be analyzed, modified and personalized.

Mathematica is a commercial software environment with a large library of primitive functions tuned for mathematical exploration, symbolic calculation, and visualization. Version 6 has three new features worth noting: (1) When a 3D graphic expression is evaluated, it appears in a form allowing virtual trackball rotation, zooming, and panning. It is easy to spin an object and get a sense of its 3D form. (2) The *Manipulate* function creates sliders which control parameters of an expression. When the sliders are moved, the expression is automatically recalculated and the new result is displayed. By putting a graphic expression in a *Manipulate* command, parameters can be interactively adjusted to obtain a desired visual appearance. (3) A library of polyhedra is available that can be used as the seeds for various algorithmic explorations.

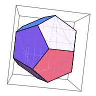
On the negative side, *M* is expensive. I would not have used it if my university did not have a site license giving students inexpensive access. Another issue is that *M* is a functional programming environment designed around list operations. Students accustomed only to object-oriented procedural programming need time to adapt to new ways of thinking. This is a good educational opportunity, however teachers planning a similar course need to plan time for students to adjust.

The examples below are downloadable from [2], with comments omitted here to save space. Each can be pasted in to *M* to get the output shown. In class, students went beyond these examples, designing their own objects, making mathematical models of them, writing software to view and adjust the forms, producing *stl* files, and building physical copies on SFF machines to keep. This paper benefits from much student input.

Examples

1. Built-in Polyhedra *M*'s polyhedron data base includes Platonics, Archimedeans, and their duals. The function `PolyhedronData[]` takes a string and returns the polyhedron with that name. It is returned in a form called a "graphics complex" (GC) which displays as a resizable 3D image that can be rotated by dragging the mouse over it. Here is an example of a line typed into M and, at right, the output it displays:

```
PolyhedronData["Dodecahedron"]
```



The GC format contains various kinds of information, so we write functions to extract the data we need. The details of these functions are not important to understand. They simply access parts of *M*'s (arbitrary) GC format. For example, the xyz coordinates of the vertices of a GC are in the position subscripted as [[1,1]], so are extracted by the function: `vertices[gc_]:= N[gc[[1,1]]]` *M*'s syntax for defining functions uses ":=" and the trailing underbar after the variable `gc` is how we declare it to be a parameter. The `N[]` function causes the result to be returned in a numeric (floating point) format, without symbolic expressions such as square roots. Below, we use this function to find the vertices of the built-in cube and observe that it is an axis-aligned unit-edge-length cube centered at the origin. The output, at right, is a list of eight points; each point is a list of the form {x,y,z}.

```
                                    {{-0.5,-0.5,-0.5}, {-0.5,-0.5,0.5},
vertices[PolyhedronData["Cube"]]     {-0.5,0.5,-0.5}, {-0.5,0.5,0.5},
                                     {0.5,-0.5,-0.5}, {0.5,-0.5,0.5},
                                     {0.5,0.5,-0.5}, {0.5,0.5,0.5}}
```

The faces of a GC can be extracted with an analogously obscure conversion function:

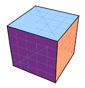
```
faces[gc_] := Map[vertices[gc][[#]]&, gc[[1,2,1]],{2}]
```

Each face is a list of points recorded in counterclockwise order as seen from outside the object, so the list of faces is a list of lists of lists. The generality of the list data type is convenient because many built-in list operations can be used on lists with any type of content. However, the programmer must be careful to keep track of what each list represents. Typing this assignment to the variable `cube` generates the six lines of output at right, which we understand to represent a cube as having six faces, each with four points:

```
cube = faces[PolyhedronData["Cube"]]
            {{{0.5,0.5,0.5},{-0.5,0.5,0.5},{-0.5,-0.5,0.5},{0.5,-0.5,0.5}},
             {{0.5,0.5,0.5},{0.5,-0.5,0.5},{0.5,-0.5,-0.5},{0.5,0.5,-0.5}},
             {{0.5,0.5,0.5},{0.5,0.5,-0.5},{-0.5,0.5,-0.5},{-0.5,0.5,0.5}},
             {{-0.5,0.5,0.5},{-0.5,0.5,-0.5},{-0.5,-0.5,-0.5},{-0.5,-0.5,0.5}},
             {{-0.5,-0.5,-0.5},{-0.5,0.5,-0.5},{0.5,0.5,-0.5},{0.5,-0.5,-0.5}},
             {{-0.5,-0.5,0.5},{-0.5,-0.5,-0.5},{0.5,-0.5,-0.5},{0.5,-0.5,0.5}}}
```

We will use this list-of-faces format in the operations below without printing it out. Unlike a GC, it does not display as a mouse-rotatable polyhedron. So we write a viewing function which converts our list-of-faces format into a GC for display. We use an option to omit the default bounding box, which can be seen above around the dodecahedron. Then the view command shows our cube. In *M*, we can rotate it to see all sides:

```
view[obj_] := Graphics3D[Map[Polygon,obj], Boxed→False]
view[cube]
```



To physically produce the objects in this paper, we can convert any list-of-faces `object` into the .stl file format for a rapid prototyping machine with a command like `Export["cube.stl", view[object]]`.

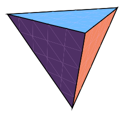
2. Tetrahedron and compounds The following function tests if a point is above a given triangle, i.e., neither in the triangle's plane nor below it. It returns *True* or *False* and is used as a test in definitions below. The triangle is given by its three vertex points. The *Module* syntax lets us define the face normal, a local variable, as the cross product of two edges. It works by using a dot product (".") to project the point in the direction of the normal and comparing that to the projection of a vertex in the same direction.

```
above[p_,{v0_,v1_,v2_}] := Module[{normal=Cross[v1-v0,v2-v1]},(p-v0).normal>0]
```

To distinguish an object's interior from it's exterior, SFF machines require that vertices of each face be listed in counterclockwise order. A function to create a tetrahedron from a list of four vertices determines this ordering from a single test of aboveness, using M's *If* [*test, then, else*] syntax:

```
tetra[{p1_,p2_,p3_,p4_}]:=
  If[above[p1,{p2,p3,p4}], {{p1,p2,p3},{p1,p3,p4},{p1,p4,p2},{p3,p2,p4}},
                           {{p1,p3,p2},{p1,p4,p3},{p1,p2,p4},{p3,p4,p2}}]
```

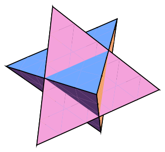
As an example, we define v4 to be a list of four nonadjacent vertices of a cube so we can build a regular tetrahedron from them:



```
v4={{1,1,1},{1,-1,-1},{-1,1,-1},{-1,-1,1}}
view[tetra[v4]]
```

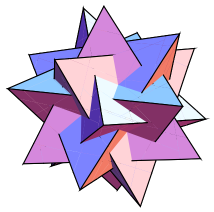
The *Join* function combines two lists into one. A minus sign applied to a list distributes to the elements of the list. So the following line produces the faces for the "Stella Octangula," i.e., the compound of two tetrahedra in a common cube. (Faces pass through each other, but this will build on most SFF machines.)

```
view[Join[tetra[v4], tetra[-v4]]]
```



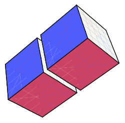
Next we form the uniform compound of five tetrahedra. Let v be the vertices of a regular dodecahedron. Five different subsets of four vertices are joined to give the following. The method (a hand drawing) to identify the vertices in each tetrahedron, is not shown.

```
v=vertices[PolyhedronData["Dodecahedron"]]
```



```
view[Join[tetra[{v[[1]],v[[11]],v[[17]],v[[20]]}],
          tetra[{v[[2]],v[[4]],v[[7]],v[[9]]}],
          tetra[{v[[3]],v[[12]],v[[13]],v[[15]]}],
          tetra[{v[[5]],v[[8]],v[[14]],v[[18]]}],
          tetra[{v[[6]],v[[10]],v[[16]],v[[19]]}]]]
```

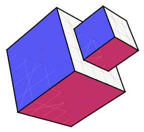
3. Transformations: translate, scale, and rotate We translate an object by adding a vector offset to each vertex. The syntax used here is a bit cryptic, with *Map* used to create a list by applying a function to elements of a given list, the "&" symbol used to create an anonymous function with "#" as the argument placeholder, and "{2}" as a special argument to *Map* indicating the function is to be applied at "level 2" (the xyz points) rather than to the top level elements of the list (the faces). We test the function by creating a cube with another cube translated 1.1 units in the X direction.



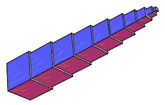
```
translate[obj_, offset_]:= Map[(#+offset)&, obj, {2}]
view[Join[cube, translate[cube,{1.1,0,0}]]]
```

We scale an object by multiplying it by a scale factor. Multiplication of a scalar times a list distributes into the list, so eventually percolates down to the individual X, Y, and Z coordinates of the points in the faces in the object. Note that an "*" is not required to indicate multiplication in *M*'s syntax; a space will suffice. So the following generates a unit cube and a translation of a half-size cube.

```
view[Join[cube, translate[0.5 cube, {0.75,0,0}]]]
```



M's syntax for creating the list {*f*[1], *f*[2], …, *f*[10]} is *Table*[*f*[*i*], {*i*, 1, 10}]. So we can create a ten-layer "wedding cake," joining cubes of size 1 through size 10, as follows. The effect of *Apply*[*Join*, *X*]] is to join all the lists in *X* into one list.

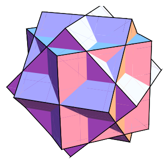


```
view[Apply[Join, Table[translate[i cube,{i^2/2,0,0}], {i,1,10}]]]
```

We rotate an object by multiplying its vertex coordinates by a rotation matrix. There is a built-in function that creates the rotation matrix corresponding to a given angle and rotation axis. M. C. Escher's *Waterfall* shows a compound of three concentric cubes. They derive from three copies of a single cube, rotated 45 degrees about the X, Y, and Z axes respectively. We can build it as follows:

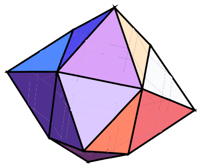
```
rotate[obj_, angle_, axis_] :=  Map[(RotationMatrix[angle,axis].#)&, obj, {2}]
```

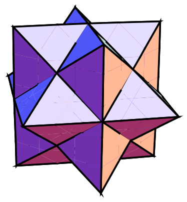
```
view[Join[rotate[cube, Pi/4, {1,0,0}],
          rotate[cube, Pi/4, {0,1,0}],
          rotate[cube, Pi/4, {0,0,1}]]]
```



4. Poke By “poke” we mean a function to create a pyramid on each face of a given object. (This is different from Kepler’s *stellation* operation, which extends the face planes.) The following code is broken down into: (1) *unit*, which produces a normalized (unit length) vector; (2) *average*, which computes the mean of the elements in a given list, and is used to find the center of a face as the average of its vertices; (3) *pokeface*, which returns a set of triangles that meet at an apex some given height above the center of one face; and (4) *poke*, which combines the triangles obtained by poking each of the faces. *M*’s *Module* syntax is used to declare *apex* and *face1* as local variables.

```
unit[vec_]:=vec/Sqrt[vec.vec];
average[L_]:=Apply[Plus,L]/Length[L];
pokeFace[face_, height_]:=Module[{apex,face1},
  apex=average[face]+
    height*unit[Cross[face[[1]]-face[[2]],face[[2]]-face[[3]]]];
  face1=Append[face,face[[1]]];
  Table[{face1[[i]], face1[[i+1]],apex}, {i,1,Length[face]}] ];
poke[obj_,height_]:=Apply[Join, Map[(pokeFace[#,height])&,obj]]
```





For example, we apply *poke* to our cube, making pyramids of height 0.7:

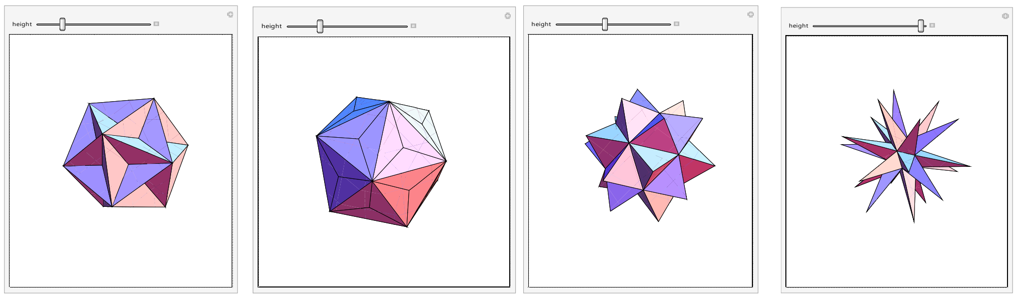
```
view[poke[cube, 0.7]]
```

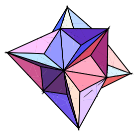
Escher’s *Waterfall* includes a poked rhombic dodecahedron as well:

```
view[poke[faces[PolyhedronData["RhombicDodecahedron"]], 0.8]]
```

For interactively changing parameters, the *Manipulate* function creates sliders and re-evaluates its arguments when you drag the slider control. (Many interesting *Manipulate* examples are available online at Wolfram’s Demonstration Project [1].) By adjusting the slider, this line of code changes the poke height, interactively giving the following forms. In the first, the height is negative; in the second, it is zero.

```
Manipulate[view[poke[faces[PolyhedronData["Icosahedron"]], height]],
           {height,-1,3}]
```

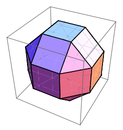




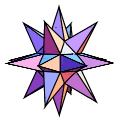
Functions can be nested to produce many new forms. Here, we poke a poked cube:

```
view[poke[poke[cube,1], -.15]]
```

Using “function overloading,” we next create a *poke* function with a third argument, *n*, that controls which faces are poked—only *n*-sided faces are poked. For example, consider the small rhombicuboctahedron, which has both 3-sided and 4-sided faces. If we poke only the 4-sided faces, we get a common form of the “Moravian Star” in which the eight original triangles receive no pyramid:

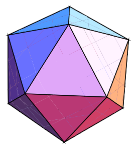


```
poke[obj_,height_,n_] := Apply[Join, Map[
                (If[Length[#]==n, pokeFace[#,height], {#}])&, obj]]
PolyhedronData["SmallRhombicuboctahedron"]
view[poke[faces[PolyhedronData["SmallRhombicuboctahedron"]], 2, 4]]
```



5. Convex Hull The convex hull of a set of points is the smallest convex body containing all the points. Finding the hull of a given set of points is a building block in more complex algorithms. Here we use a simple quadratic-time iterative algorithm described in many texts, e.g., [3]. First create a tetrahedron from four of the points. Then consider each remaining point in turn, see which existing faces it is above, remove them, and connect the unmatched edges to the point. The internals of the algorithm are not crucial on first reading.

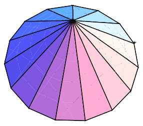
```
edgesOfTriangle[{v0_,v1_,v2_}] := {{v0,v1},{v1,v2},{v2,v0}}

remove[{},x_] := {};
remove[L_,x_] := If[First[L]==x, Rest[L], Prepend[remove[Rest[L],x],First[L]]]

unmatchedEdges[List_] := Module[{},
  Clear[unmatched];
  unmatched[x_] := True;
  Scan[(unmatched[Reverse[#]]=False)&, List];
  Select[List, unmatched]
  ]

cHull[L_] := Module[{faces,keepers,edges,newTriangles},
  If[Length[L]<4,Print["cHull needs at least 4 points"]];
  faces=tetra[Take[L,4]];
  For[i=5,i<=Length[L],i++,(
    keepers=Select[faces,(!above[L[[i]],#])&];
    edges=Apply[Join, Map[edgesOfTriangle, keepers]];
    newTriangles=Map[({#[[2]],#[[1]],L[[i]]})&, unmatchedEdges[edges]];
    faces=Join[keepers, newTriangles];
    )];
  faces
  ]
```



It creates a triangulated polyhedron from a list of its vertices:

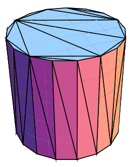
```
view[cHull[vertices[PolyhedronData["Icosahedron"]]]]
```

We can use the hull function to define a function that creates an *n*-sided pyramid. Just hull *n* points equally spaced around a circle and one point above the center:



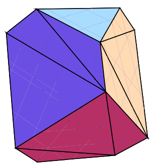
```
pyramid[n_]:=cHull[Prepend[Table[{Sin[2 Pi i/n],Cos[2 Pi i/n],0},
                                 {i,1,n}], {0,0,1}]//N];
view[pyramid[15]]
```

Similarly, an *n*-sided prism (with triangulated faces) is generated as the hull of the points around two circles. (Here we use *RandomSample* to shuffle the points, because our simple *cHull* method may fail if the first five points are in a common plane.)



```
prism[n_]:=cHull[RandomSample[Apply[Join,
   Table[{Sin[2 Pi i/n],Cos[2 Pi i/n],z},{i,0,n-1},{z,-1,1,2}]//N]]];
view[prism[17]]
```

As an art example, we can create (a triangulated) Durer’s polyhedron [4] from its vertices.



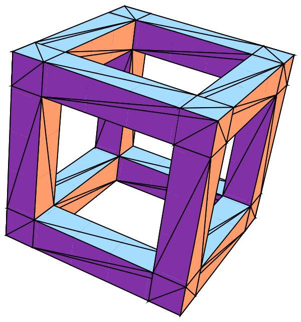
```
durerPoints={{-0.70711,-0.40825,-0.44174},{-0.70711,0.40825,0.44174},
             {-0.43702,-0.25231,-0.7792},{-0.43702,0.25231,0.7792},
             {0.,-0.8165,0.44174},{0.,-0.50462,0.7792},
             {0.,0.50462,-0.7792},{0.,0.8165,-0.44174},
             {0.43702,-0.25231,-0.7792},{0.43702,0.25231,0.7792},
             {0.70711,-0.40825,-0.44174},{0.70711,0.40825,0.44174}};

view[cHull[durerPoints]]
```

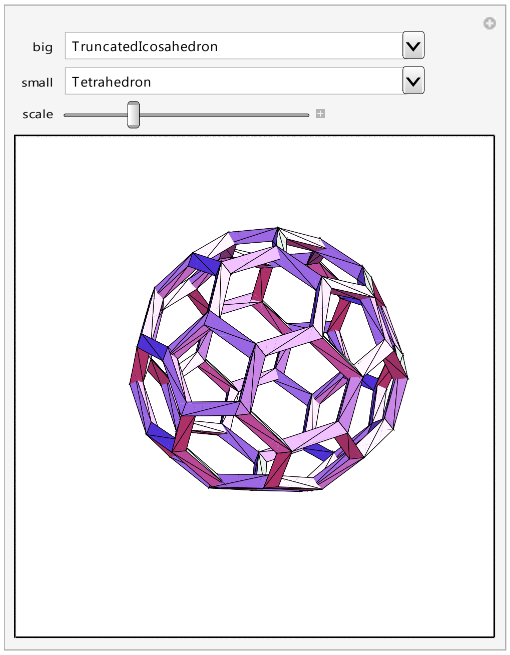
6. Solid-edge models As another use of the convex hull, again skimming over details, consider this simple algorithm for making an "edge model" of a polyhedron: Imagine a small polyhedron centered on each vertex of a large polyhedron. For each edge of the large polyhedron, generate a strut by taking the convex hull of the vertices of the two corresponding small polyhedra. Joining all the struts gives a Leonardo-style open faced polyhedron. The volume around each vertex is overlapped by all the struts that meet there.

```
edgeIndicesOfFace[face_] := Select[Transpose[{face,RotateLeft[face]}],
                                   (#[[1]]<#[[2]])&];
edgeIndices[gc_] := Apply[Join, Map[edgeIndicesOfFace, gc[[1,2,1]]]];
edges[gc_] := Map[vertices[gc][[#]]&, edgeIndices[gc],{2}];
translateVertices[name_,scale_,xyz_] := Map[(xyz+#)&,
                                         scale vertices[PolyhedronData[name]]];
strut[name_,scale_,edge_]:=cHull[Join[translateVertices[name,scale,edge[[1]]],
                                      translateVertices[name,scale,edge[[2]]]]]
edgeModel[name_,nameVertex_,scale_] := Apply[Join,
                  Map[strut[nameVertex,scale,#]&, edges[PolyhedronData[name]]]]
```

For example, we make an open cube using a small (0.2 size) cube at each vertex (shown below).

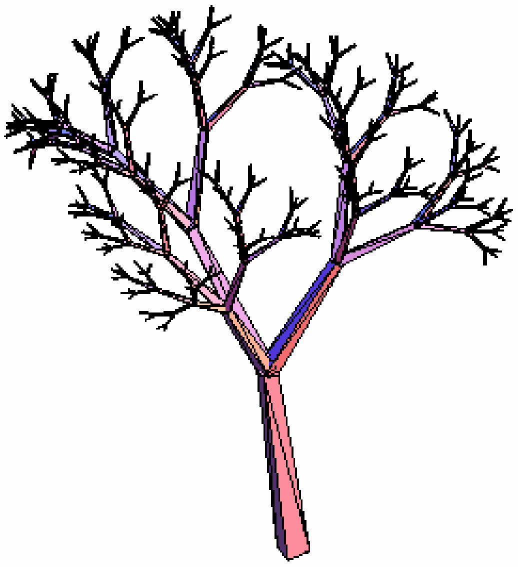
```
view[edgeModel["Cube","Cube",0.2]]
```




big: TruncatedIcosahedron
small: Tetrahedron

For trying out various scale factors and different polyhedra for the vertices, *Manipulate* is natural here. The idiom *PolyhedronData*[*All*] gives a list of all available polyhedra which we use as options in a drop-down box. In the example shown, we make a truncated icosahedron (soccer ball) using tetrahedra for each corner, as they result in simple triangular struts.

```
Manipulate[view[edgeModel[big,small,scale]],
   {big, PolyhedronData[All]},
   {small, PolyhedronData[All]},
   {scale, 0.001, 1}]
```



7. Fractal Tree As a simple example of recursion, we generate a tree where each branch is a scaled down tree of the same form. A branch from point A to point B is produced by taking the convex hull of a small square around A and a small square around B. The size of each square is chosen proportionally to the distance between the points, to maintain a constant aspect ratio, and the upper square is smaller, for a taper. A rotated frame of reference is created for each sub-tree. *M*'s `Reap` and `Sow` mechanism collects the branches into a single list.

```
branch[p1_,p2_] := Module[{v1,v2,size=0.08 Sqrt[(p1-p2).(p1-p2)],t=.6},
  v1=size unit[Cross[(p2-p1),{1,2,3.4567}]];
  v2=size unit[Cross[(p2-p1),v1]];
  cHull[{p1+v1, p1-v1, p1+v2, p1-v2,
         p2+t v1, p2-t v1, p2+t v2, p2-t v2}]]
```

```
tree[level_,origin_,frame_,size_] := Module[{top, M},
  If[level==0,Return[]];
  top=origin+size frame[[3]];
  Sow[branch[origin,top]];
  tree[level-1,top,
       RotationMatrix[Pi/5,{1,0,0}].frame,.6 size];
  tree[level-1,top,
       RotationMatrix[Pi/6,{-.5,.6,0}].frame,.7 size];
  tree[level-1,top,
       RotationMatrix[Pi/7,{-.5,-.6,0}].frame,.5 size];]
```

```
makeTree[level_] := Reap[tree[level,{0,0,0},IdentityMatrix[3],1]][[2,1]]
```

```
view[makeTree[6]]
```

8. Fractal Polyhedron The next example generates a wide variety of fractal polyhedra. For each significant point (vertex, edge midpoint, or face center) of a polyhedron, place a smaller (scaled by alpha) copy of the same set of points, down to a stopping level at which we put a small (scaled by beta) polyhedron of another type, possibly stellated. The code is too dense to explain in detail, but is included to inspire the reader to investigate further:

```
points[name_] := With[{gc=PolyhedronData[name]},
                {vertices[gc], Map[average,edges[gc]], Map[average,faces[gc]]}];
```

```
allSums[L1_,L2_]:=Apply[Join,Outer[Plus,L1,L2,1]]
```

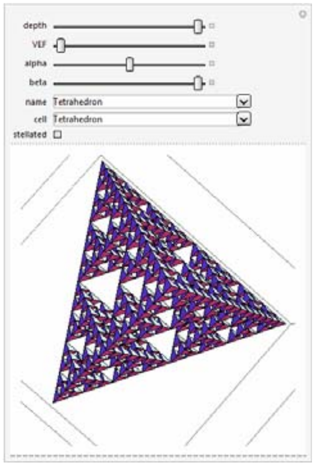
```
centers[depth_,type_,alpha_,points_]:= If[depth==0,{{0,0,0}},
          allSums[points[[type+1]],alpha centers[depth-1,type,alpha,points]]];
```

```
translates[obj_,xyzs_]:=Apply[Join,Map[translate[obj,#]&,xyzs]];
```

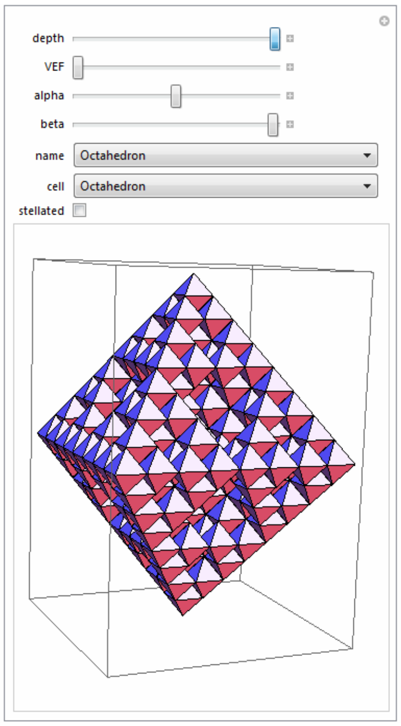
```
Needs["PolyhedronOperations`"];
```

```
makeCell[stell_,name_]:=faces[If[stell, Stellate[PolyhedronData[name],2.83],
                                        PolyhedronData[name]]];
```

```
Manipulate[Graphics3D[Map[Polygon, translates[(beta alpha^(depth-1))
           makeCell[stellated,cell], centers[depth,VEF,alpha,points[name]]]]],
   {depth,0,3,1}, {VEF,0,2,1}, {{alpha,.1},0,1}, {{beta,.1},0,1},
   {name,PolyhedronData[All]}, {cell,PolyhedronData[All]},
   {stellated,{False,True}}]
```


depth
VEF
alpha
beta
name Tetrahedron
cell Tetrahedron
stellated ☐

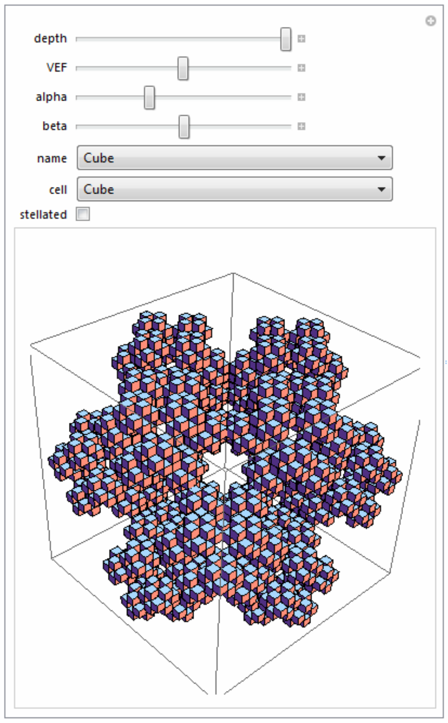
This generalizes the simple special case (at right) of the well-known Sierpinski tetrahedron. In the first example below, we make an octahedron with smaller octahedra at each vertex. In the second, we make a cube with a smaller cube at each edge midpoint. In the third, we make a rhombic dodecahedron with a smaller stellated rhombic dodecahedron at each face center.



name Octahedron

cell Octahedron

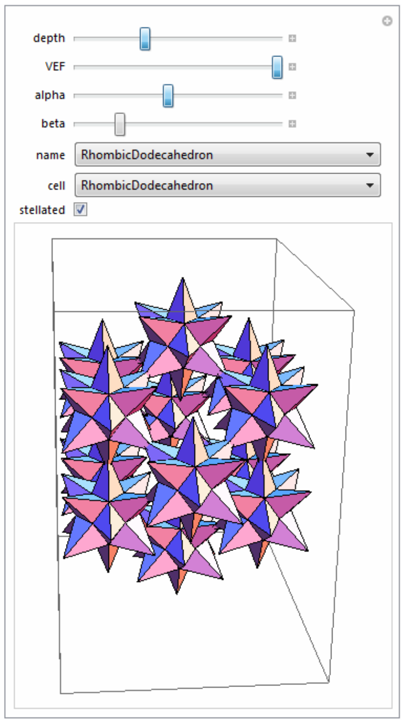
stellated ☐



name Cube

cell Cube

stellated ☐



name RhombicDodecahedron

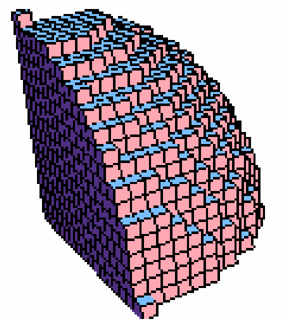
cell RhombicDodecahedron

stellated ☑

9. Marching Cubes The "Marching Cubes" algorithm generates a boundary representation from a Boolean function of space. As simply implemented here, *f* is a Boolean function of {x,y,z}; upper and lower bounds are given for X, Y, and Z; and a step-size d specifies the level of detail. The algorithm scans through a lattice of points, looking for points where the Boolean value of the function is opposite that of a neighboring point, and creates a square separating those adjacent voxels.

```
boundary[f_,xmin_,xmax_,ymin_,ymax_,zmin_,zmax_,d_] := Module[{g},
  g[x_,y_,z_]:=If[xmin <= x <= xmax && ymin <= y <= ymax &&
                zmin <= z <= zmax, f[x,y,z], False];
  Reap[
    For[x=xmin,x<=xmax,x+=d,
     For[y=ymin,y<=ymax,y+=d,
      For[z=zmin,z<=zmax,z+=d,
       If[g[x,y,z],(
        If[!g[x+d,y,z],Sow[{{x+d,y,z},{x+d,y+d,z},{x+d,y+d,z+d},{x+d,y,z+d}}]];
        If[!g[x-d,y,z],Sow[{{x,y,z},{x,y,z+d},{x,y+d,z+d},{x,y+d,z}}]];
        If[!g[x,y+d,z],Sow[{{x,y+d,z},{x,y+d,z+d},{x+d,y+d,z+d},{x+d,y+d,z}}]];
        If[!g[x,y-d,z],Sow[{{x,y,z},{x+d,y,z},{x+d,y,z+d},{x,y,z+d}}]];
        If[!g[x,y,z+d],Sow[{{x,y,z+d},{x+d,y,z+d},{x+d,y+d,z+d},{x,y+d,z+d}}]];
        If[!g[x,y,z-d],Sow[{{x,y,z},{x,y+d,z},{x+d,y+d,z},{x+d,y,z}}]]; )]
      ]]]][[2,1]]]
```

As an example, we define a function which is *True* within a sphere of radius 15 and use marching cubes to build a low-resolution boundary around it, in the positive octant of space between 0 and 20:

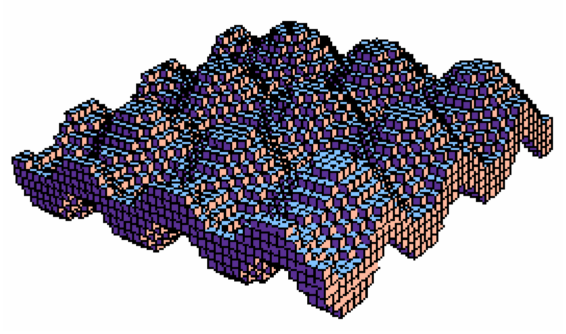


```
fSphere[x_,y_,z_] := x^2+y^2+z^2 <= 15^2
view[boundary[fSphere, 0,20, 0,20, 0,20, 1]]
```

As a second example, create a function which is true within a small distance of an “egg-carton surface” in space, and make a low-resolution boundary for it.

```
fWave[x_,y_,z_]:=Abs[Sin[x]+Sin[y]-z]<1;
```

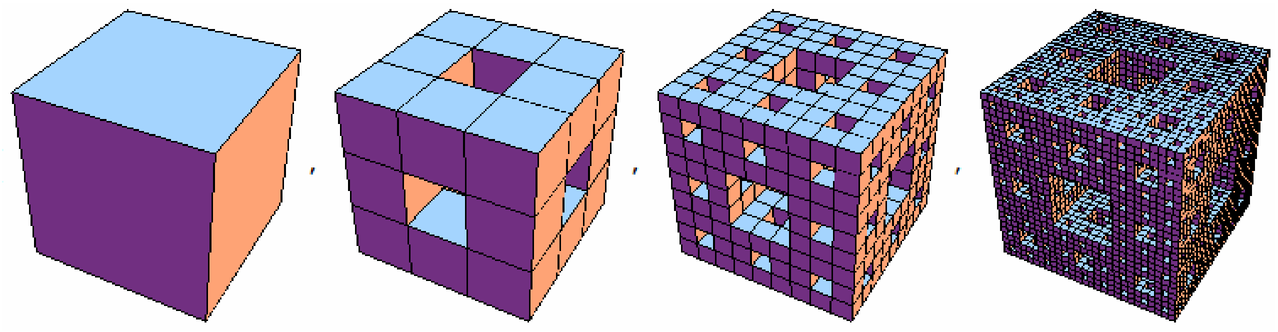
```
view[boundary[fWave, 0,20, 0,20, -3,3, .5]]
```



10. Menger Sponge The Menger Sponge function is true at those integer lattice points which, when expressed in base 3, do not have two (or more) of their three
(X,Y,Z) coordinates 1 in the same trit position. The Marching Cubes algorithm then produces its boundary. To the right of the *M* images below is a photograph of a SFF model generated from the following program.

```
fMenger[i_,j_,k_]:=Module[{id=IntegerDigits[i,3,7], jd=IntegerDigits[j,3,7],
                           kd=IntegerDigits[k,3,7]},
  For[n=1, n<=7, n++,
   If[(id[[n]]==jd[[n]]==1) || (id[[n]]==kd[[n]]==1) || (jd[[n]]==kd[[n]]==1),
    Return [False]]];
  Return[True]]
```

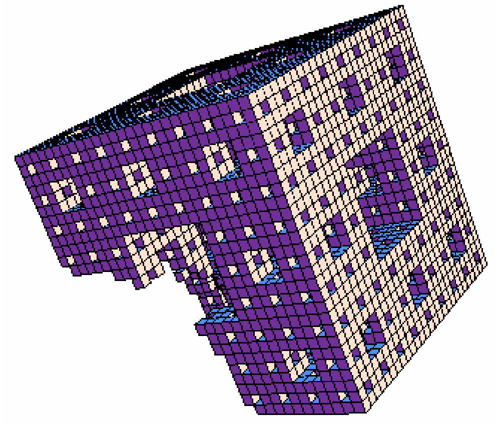
```
Table[(max=3^depth-1;view[boundary[fMenger,0,max,0,max,0,max,1]]), {depth,0,3}]
```



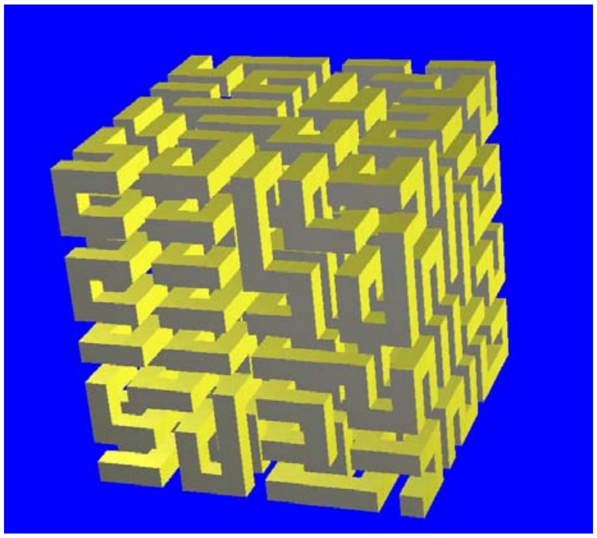
11. Constructive Solid Geometry The Boolean *And* or *Or* connectives applied to Boolean functions of space give us functions for the intersection or union of the corresponding objects. So it is easy to find any Boolean combination of objects in voxel format. For example, to subtract the above sphere from the Menger sponge, we create a difference function which is true for points that are in the sponge and not in the sphere. This generalizes easily to all unions, intersections, differences, and complements.

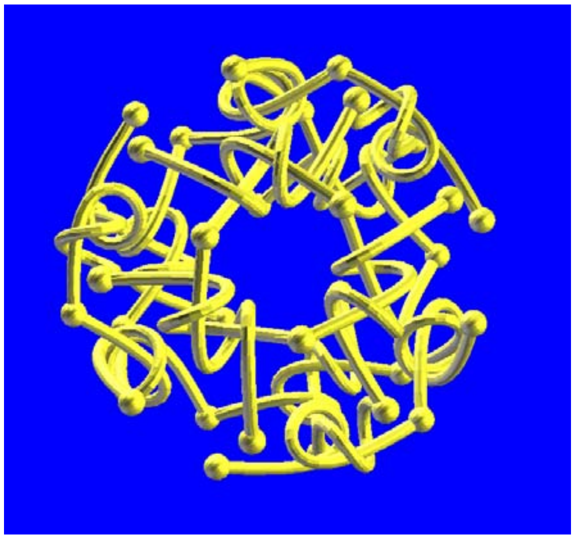
```
fDiff[x_,y_,z_] := And[fMenger[x,y,z],
                       Not[fSphere[x,y,z]]];
```

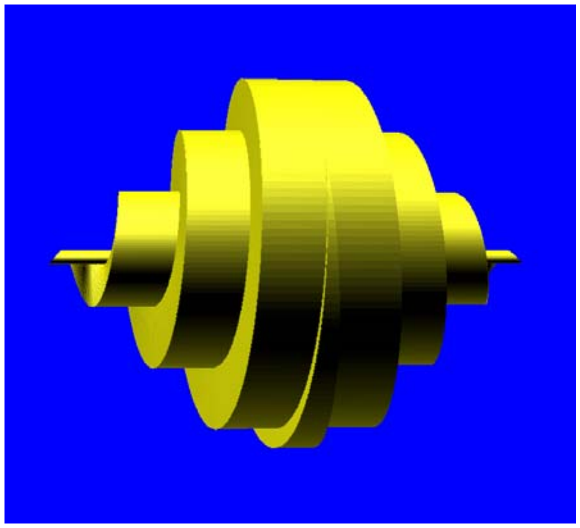
```
view[boundary[fDiff, 0,26, 0,26, 0,26, 1]]
```



12. Student Designs After learning the above techniques, the students in the Special Topics course were able to conceive and implement their own 3D designs. These figures show renderings of several examples.







Conclusions

This intensive survey illustrates techniques of interest to those who may want to teach a course or learn on their own about 3D design with *Mathematica*. Each example can be studied, modified, and extended. Interactive design with the *Manipulate* function makes this fun and easy, but can only be hinted at with the still images in this paper. Space does not permit inclusion of artworks to motivate these examples. But in a classroom, each example can be introduced with a discussion of the data structures involved, the algorithmic issues, and artwork such as Escher's *Waterfall*, Moravian stars, Durer's *Melancolia I*, Leonardo da Vinci's drawings of polyhedra, or modern geometric sculpture, to put this work in a fuller context.

References

[1] Mathematica, `http://www.wolfram.com`
[2] George W. Hart, `http://www.georgehart.com`
[3] Joseph O'Rourke, *Computational Geometry in C*, Cambridge University Press, 1998
[4] Eric Weisstein, "Durer's Solid," `http://mathworld.wolfram.com/DuerersSolid.html`