Tools for designing programmatic self-assembling systems.

Saul Griffith.

6.978 final paper, Dec 2002

"...until an adequate theory of automata exists there is a limit to the complexity and capacity of the automata we can fabricate".

– Von Neumann.

Abstract

It is of interest to add more control and complexity to the processes of self-assembly that we are just beginning to understand. Existing self-assembling systems are essentially meso or macro-scale versions of crystallization. I wish to expand the tool-box of self-assembly to include dynamic components that emulate the behaviour of allostery exhibited in bio-macromolecules. An allosteric molecule can be considered abstractly as a state machine where each stable conformation is a state. Designing the components for self-assembly gets increasingly complicated as part numbers increase, and as the components have re-configurable 'states'. Essentially the problem in designing self-assembling components is to avoid undesirable meta-stable states, and to make the desired assembled geometries the lowest energy conformations of the system. To that end, this work describes the development of simulation tools for the modeling, and eventual evolution of mechanical state machines for programmatic, or predicated, self-assembly.

1.Introduction.

Biology uses many examples of programmed, and self, assembly processes to achieve a remarkable array of structure and morphologies at many scales. Whilst our understanding of self assembling processes is slowly proceeding, there are no non-biological examples of on-the-fly programmable assembly processes for three dimensional structure at any scale. Sub-cellular, uni-cellular, and multi-cellular biological systems all display controlled programming of 3D structure. Commonalities between these biological systems and a "mechanical morphogenesis" are being sought. Self-assembly is oft touted as the route to building next-next generation computers with components at the nanometer scale. I would contend that without more adequate knowledge about more complex self-assembly, and quantitative rules for system information, this will not be possible.

1.Background and Goals.

1a. Self Assembly.

The goal of this work is to expand the current knowledge of ‘self-assembly’ and to introduce concepts of state and program size complexity into the components such that aperiodic and non-crystalline structures can be produced ‘automatically’.

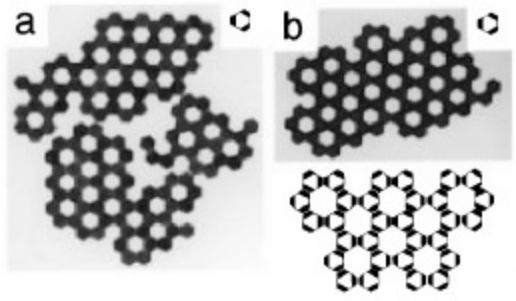

Figure 3. Regular crystalline lattices by self assembly.

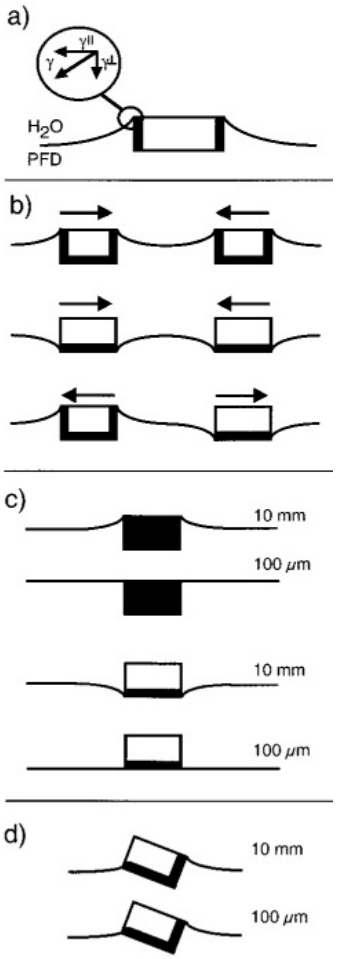

Figure 2. Attractive and repulsive forces by meniscus shape.

The physical system employed borrows from the work of Whitesides et.al. and comprises PDMS components at fluid/fluid interfaces and the minimization of surface energies to converge on an assembled state. An illustration of the forces involved and the menisci generated can be seen in figure 2. (Bowden et.al). The typical self-assembling constructions from such parts can be seen in figure 3.To date this type of self-assembling system has dominated the literature. I see it as limited in some very important ways. Firstly all such systems have significant 'program' limitations in that the individual units have a set state determined prior to assembly. In essence the only thing being programmed is the basic attractive and repulsive forces between given faces. These systems all build periodic crystalline type structures which have inherently limited complexity. In essence all of these systems are seeking out a global energy minima and hence a static equilibrium.

1b. Biology.

Biological systems do much more than this simple type of self-assembly, though crystallization and periodic arrays are also within it's toolkit. Perhaps the most important difference is that biological systems are capable of being non-equilibrium or dynamic systems, and that some biological components may also be thought of to have more than one pre-programmed state. Biology employs the allosteric function heavily in the control of self-assembly and morphogenesis, particularly in sub-cellular (bio-macro-molecule) systems (Ptashne). A schematic diagram of allostery can be seen in figure 1. The various allosteric configurations of a part can in some respects be considered states of a simple state machine. Biology also uses the concept of co-operative binding heavily, but I will not go into much detail on that in this paper.

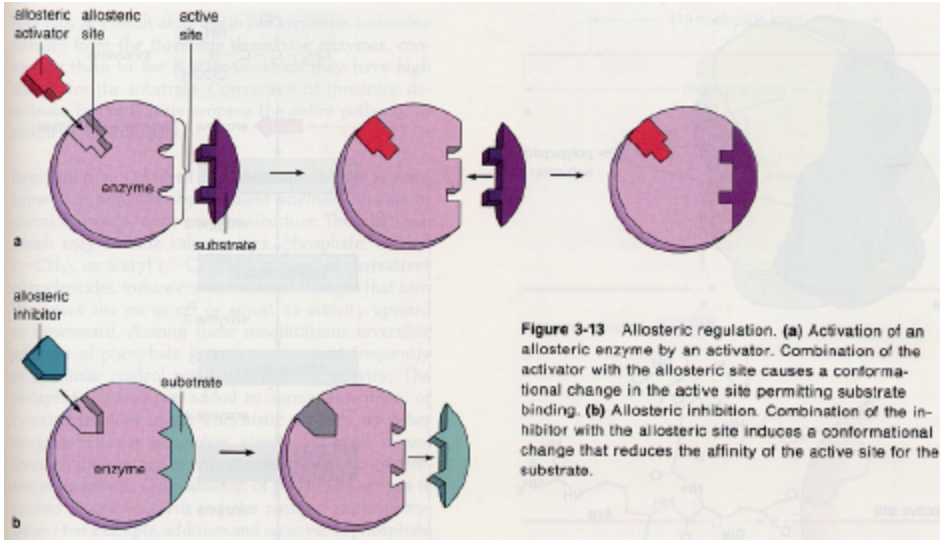

Figure 1. Schematic of allosteric 'state'. Ptashne.

By way of this introduction I have wanted to demonstrate how non-static, 'state' machine type components are an interesting topic of research in self-assembling systems.

1c. Predicated, or programmatic self-assembly and my previous work.

Where the goal of this work is to programmatically assemble 3D structure from a solution or collection of parts, two challenges can be identified.

The first is the set of questions that relate to system design: How many components of what complexity are capable of manufacturing what collection of assemblies with what complexity? How many states required at each node? What is the assembly complexity space for a given set of components? I believe we can borrow heavily from automata theory and CA's in answering these questions, however the systems to be built are inherently non-digital and the concepts of program and state as traditionally defined, are not entirely adequate.

The second set of questions relates to the individual component design: How does one build the required number of states and inter-component interactions into a unit.

I hope to make progress on questions in both of these statements, but in the particular work described in this paper I will risk putting the egg before the chicken and focus on the design of the individual components for assembly.

Every component could have a minimal micro-chip that controls electrostatic or electro-wetting interactions with other parts, and given enough memory and processor for each component an infinite number of assemblies could be produced. Power, expense, and other reasons make this an undesirable method of manufacturing our component parts, and in many ways the redundancy of having a powerful processor in every unit is un-interesting. I have chosen instead to focus on limited state machines as every unit can conceivably be manufactured in very high numbers. This is more akin to sub-cellular assembly of macro-molecules than to multi-cellular assembly.

Previous to this work I developed a 2 bit mechanical state machine that programmatically self assembles at the interface between water and poly-fluro-decalin (Figure 4). The mechanical state machine is a PDMS component with a mechanical flexure that acts as the 'switch' in the state machine. The components can be most easily envisaged as a mechanical allosteric enzyme.

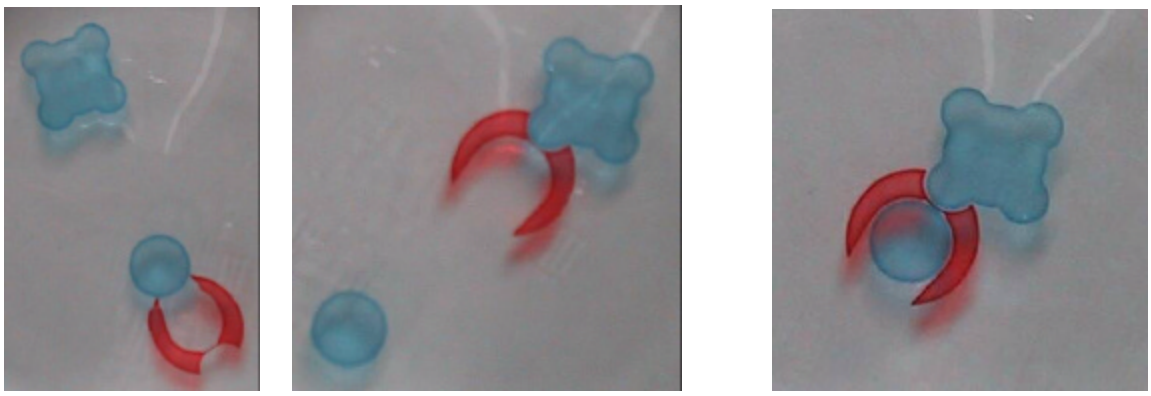

Figure 4. PDMS parts at water/PFD interface, coming together and interacting due to meniscus forces. The allosteric part is the horse-shoe shaped red component that is 'opened', or activated when the catalyst (4-lobed blue part) binds it's functional site. The bound blue circular part can only fit into the horseshoe once the catalyst is bound.

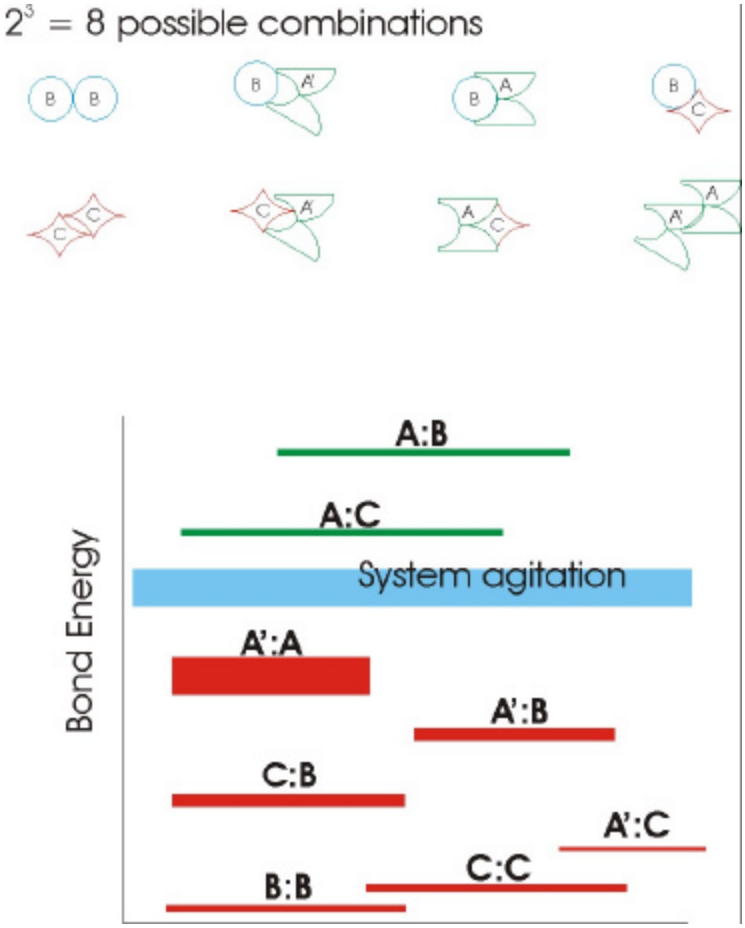

Figure 5. System energy design and component interactions.

I have quickly discovered that the extra complexity of designing state and flexible parts into the self assembling systems dramatically increases the complexity of their design. You will observe in figure 4. that there are no straight edges on the

components to avoid local energy minima on their collision. The essence of designing mechanisms for assembly in this system is designing an energy diagram of the different components geometric alignment with respect to one-another, such that the desired states sit in the deepest possible local energy minima. I have successfully done this for the 2 state system above and am making progress on a 3 state system, however it is a problem of increasing difficulty, and as more component types are added to a system the challenge is to avoid any undesirable local energy minima.

Figure 5 illustrates this point. For the example of a 3 component system where one component has two states, A & A', and the other two components single state, fixed parts, B & C, there are 8 possible combinations of parts. For each combination of parts in a 2D surface interaction there is a continuum of rotational orientations, with a subsequent bond energy relative to orientation curve. The key is to design the components such that desirable interactions A:B and A:C have stable conformations whilst the other bond energies are below the level of the system agitation energy (akin to KT in chemical systems).

I believe there is utility in having modeling tools that can facilitate this design process by automating the generation of the rotational energy diagram for any pair of components. From here it is possible to imagine using genetic algorithms or some other virtual evolutionary mechanism to evolve sets of parts with more optimal interactions and more complex interactions by performing a mechanical mutation on the part shapes at each generation. This paper describes the development of some of these tools for modeling the self-assembly of simple components at the interface of a liquid.

3.The Physical Model.

The goal for developing a working physical model of the self assembling systems I am interested in is to have an engine for designing and analyzing components and their interactions. I have employed heavily the excellent program 'Surface Evolver' of Kenneth Brakke. This program was developed for modeling soap films, mathematical geometries and solder wetting and bond formation. It uses the gradient descent method to evolve the minimal energy surface for a given and defined starting geometry. Surface Evolver may be found at: http://www.susqu.edu/facstaff/b/brakke/evolver/evolver217.html There is excellent documentation at that site and I refer those interested to read more.

3.1.Defining and importing geometry.

Surface evolver utilises a vertice, edge, face, body, hierarchy of geometric definition. 2D, 3D cartesian, 3D polar, string, and 4 and higher dimensional co-ordinate systems may be employed. Each edge, face, and body must carefully be defined within a convention such that surface normals are aligned and faces are oriented and bodies given mass. In the early part of this work I was able to

manually generate simple test geometries to execute in evolver. As part of the project I developed MATLAB code for automatically importing geometry from 3D cad programs (eg. Rhino) via the .raw (Raw Triangles) data export format. This code generates vertices, edges, and faces, in a surface evolver readable format, however the geometry files still require editing to constrain appropriate parts, and to define interfacial surface energies, and body masses. This is further work to do.

3.2.The datafile.

Surface evolver operates on a data file. This datafile includes the geometric definition of the parts, constrained geometry, contact angles between faces (a constraint), surface tensions, gravity, mass and centers of mass, optimizing parameters for running, and calls for functions (.cmd files) that implement the updating of position etc. as they file is run. An example datafile, duohex4.fe, can be found in Appendix A.

3.3 The .cmd files.

The command file which is called from the datafile is essentially a subroutine for recalculating the forces, volumes, geometric constraints, or other components of the datafile. These command files look very much like the C programming language. An example command file, xyztorque.cmd, can be found in Appendix B. In this example (xyztorque.cmd is the file called from duohex4.fe), the x,y,and z forces, and the torque, on the floating components, are calculated by the central difference method. Each physical system being modeled requires an appropriate command file which appropriately simulates the physical behaviour that one is wishing to model.

3.4 Validating the Surface Evolver physical model.

Surface evolver is typically used to model the equilibrium surfaces of, for example, a soap film on a wire frame. A lot of massaging of the datafiles and especially the command files was required to model this system. Before using the model as a predictive tool for designing components I have verified it by modeling interactions in a simple well constrained case which I can compare to physical experiments. For this simple case I am modeling the assembly of two hexagonal components on a petri dish surface with different contact angles, masses, and wetted faces.

Example 1 (figure 6) is the case of two parts which only wet on their bottom surface (top surface is not shown in this model). The components can be seen to come together iteration by iteration until their edges touch. Quantitatively the resulting minimum energy surface of the petri dish appear similar to those for mm sized hexagons at a water, air interface.

Example 2 (figure 7) is the same model modified slightly such that the sides of the hexagonal prisms also wet and hence the top face is that in contact with the air water interface. Again, the components can be seen to come together as the surface iterates. This is again reflective qualitatively of a similarly patterned system in a real dish.

Figure 8 is the same as example 1 except that the inter-component spacing has been increased. These parts do not converge as their menisci do not interact.

Figure 9 is the same example again, with the mass of the component increased beyond a point where surface tension will buoy it. It illustrates the point that the model does not rebuild the surface after the component sinks. It can be used however to model the scale vs weight issues for parts to get an idea of the instabilities in real systems.

Changing the scale of the components has similar effects such that when they are very large they converge slowly relative to their size and positions.

These examples are enough to give confidence in the physical model as representative of the physical system and therefore as a useful engine in design of more complicated parts.

Another detail of the model is illustrated well in figure 9. The facets between parts do not disappear as the parts approach each-other in the current model. In one respect this is good as it effectively does edge detection of the parts and prevents them physically overlapping. The problem is that these facets are fixed and end up pinning the parts at an orientation prematurely. In the real physical system, these two parts will converge until these two faces line up. In future iterations of the model I will regenerate these facets as they approach each-other to allow them to converge further to the expected structure.

So far I have described the generation of dynamic models that allow the user to observe the interactions as the model runs. I believe in retrospect that this is not the optimal way to generate the energy diagrams that I wish to have for one component relative to another as a function of total system energy vs. angular orientation of the two parts. To do this I have generated MATLAB code for the importing of raw triangle files from a 3D CAD program. (Appendix C). One can generate a series of these files by rotating the parts automatically within the CAD environment and upon import evolve them to an equilibrium point and sum the energy over all facets in the system to get a quantitative measure of surface energy for each orientation. These can be used to generate energy vs. theta curves. So far I can implement this manually on hand coded geometries, but have not as yet completely automated the import process.

4. Results, Conclusions.

Surface evolver was used as the scaffolding for building a physical model of self assembly at a 2D interface by meniscus forces. Using a well studied system I was able to produce qualitative results indicating that the model is valid and representational of real systems. I determined the principal limitations in the current system. In retrospect I think the dynamic model attempted is of less utility than a static model that merely evolves and measures the equilibrium energy for a system of components and their relative orientations. By running a series of these models I will have energy vs. orientation diagrams for pairwise part interactions that can either inform the manual design of the next generation of components, or in the future for the computer evolution of these parts. In summary useful tools were developed for modeling these systems and as aids to design of future mechanical implementations of the ideas outlined earlier in this paper.

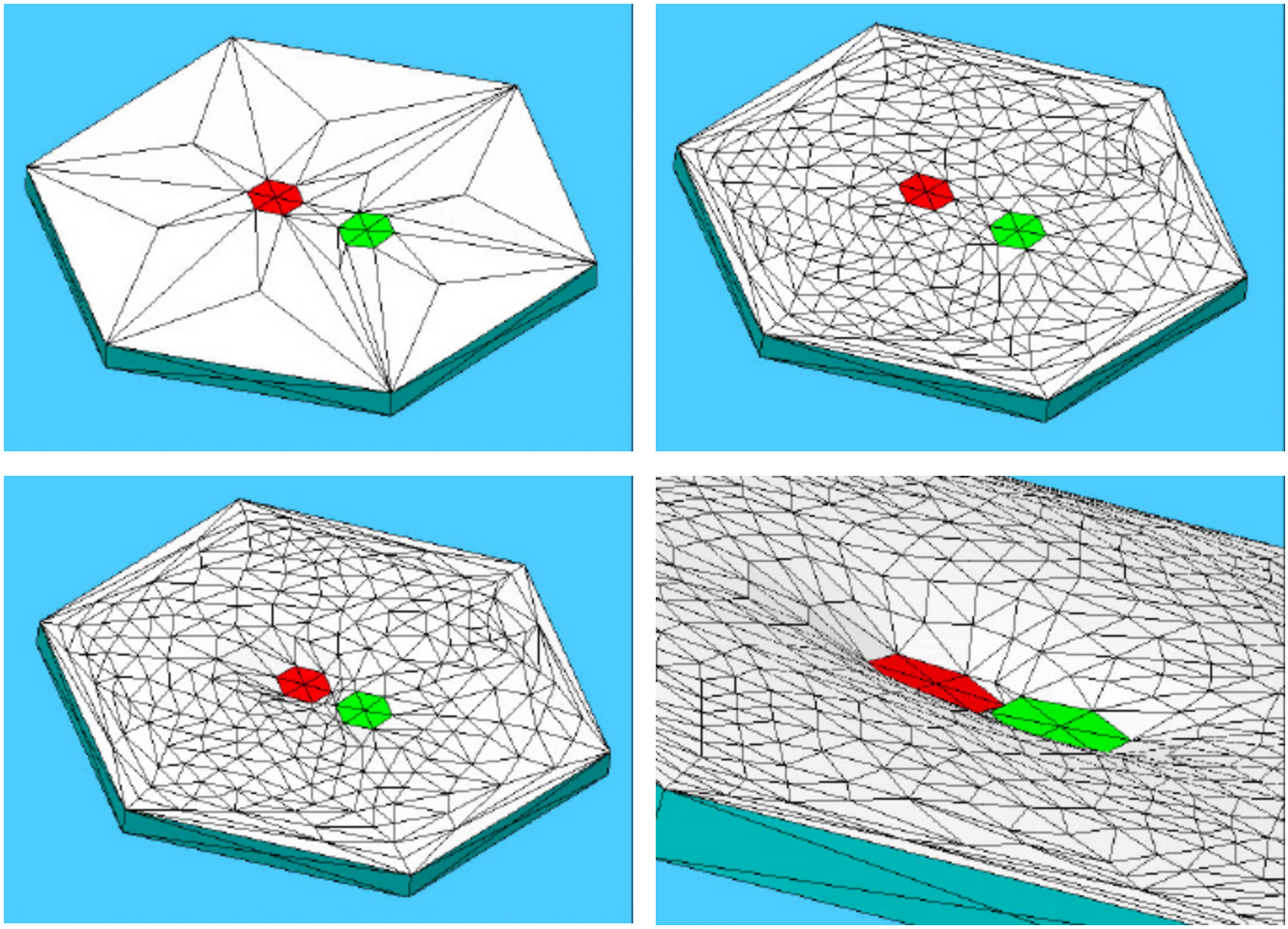

Figure 6: *Example 1: Evolution of minimum energy surface for 2 hexagons floating on an hexagonal petri dish with mass acting at their center of gravity, with one wetting face (in contact with fluid) with a contact angle of 35 degrees. Hexagons are given an initial orientation, position, and tilt angle relative to the absolute so-ordinate system. Sides and top of hexagon are not shown.*

a) Geometry of initial datafile.
b) Geometry after refinement of elements.
c) After 50 iterations of surface energy minimization.
d) Close-up of edge interactions after 120 iterations. The concave meniscus can be seen.

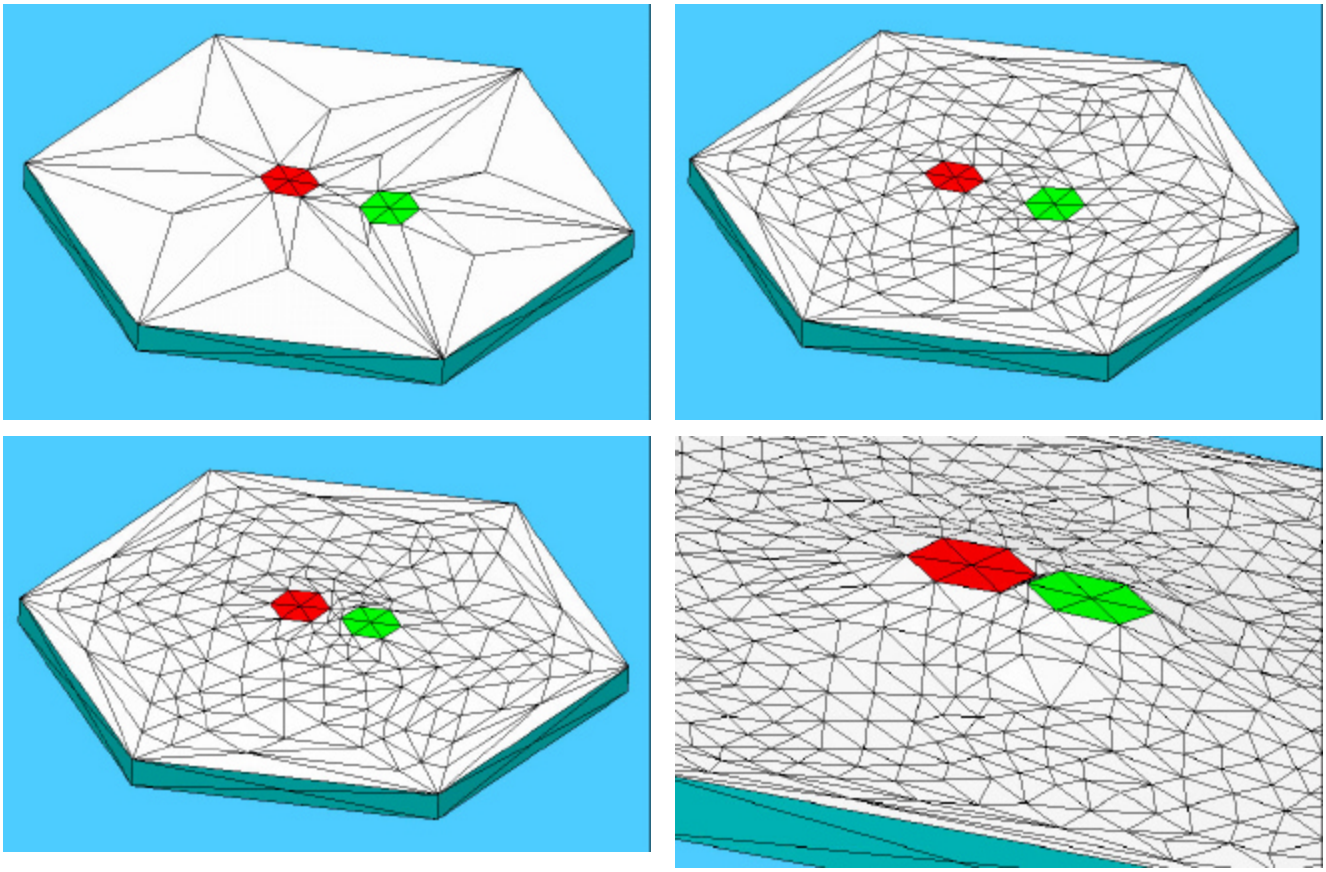

Figure 7: *Example 2: Evolution of minimum energy surface for 2 hexagons floating on an hexagonal petri dish with mass acting at their center of gravity, with wetting sides and bottom surface (in contact with fluid) with a contact angle of 65 degrees. Hexagons are given an initial orientation, position, and tilt angle relative to the absolute so-ordinate system.*

a) *Geometry of initial datafile.*
b) *Geometry after refinement of elements.*
c) *After 50 iterations of surface energy minimization.*
d) *Close-up of edge interactions after 120 iterations. In this case the convex meniscus can be seen.*

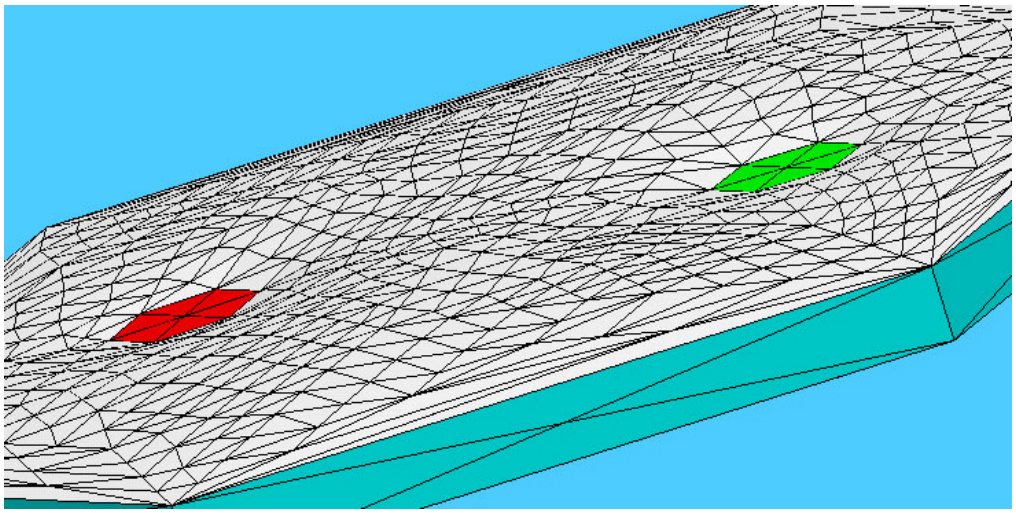

Figure 8. Non converging hexagons at an interface due to large separation distance.

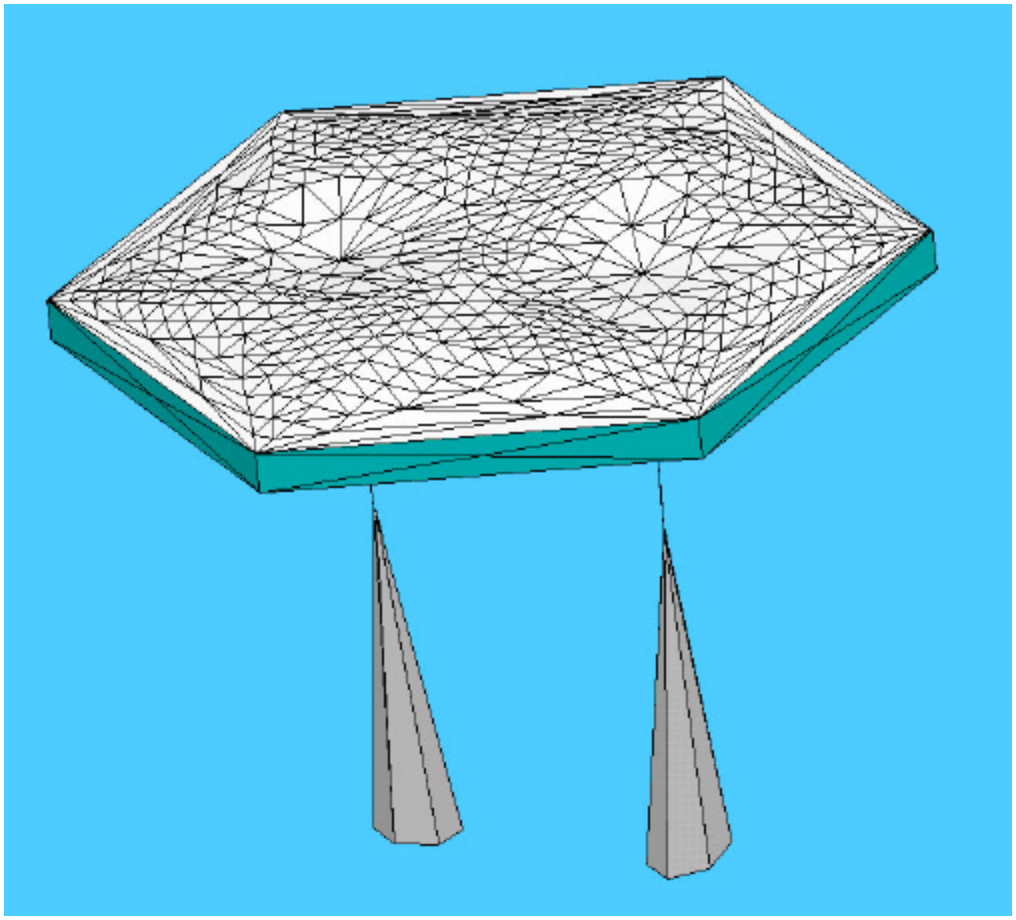

Figure 9. When the mass of the floating objects is too high, or they become too large, they sink into the bath. The model does not recaculate the reformation of a clean surface once the parts have fallen through.

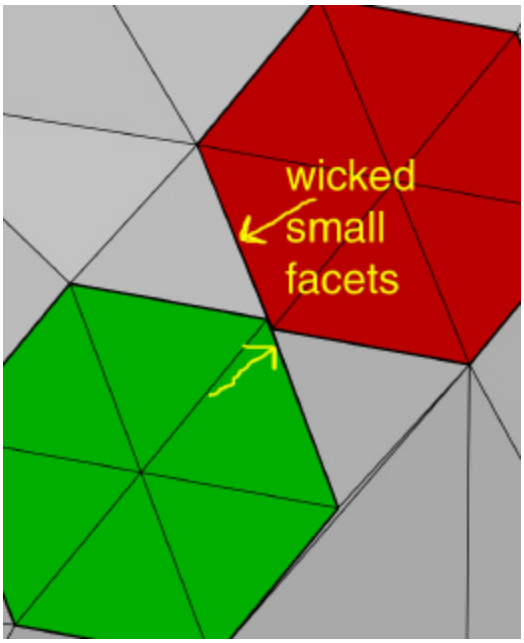

Figure 9: Faces do not quite meet as small facets remain and are squeezed between edges. This effectively pins the surfaces short of their real equilibrium position.

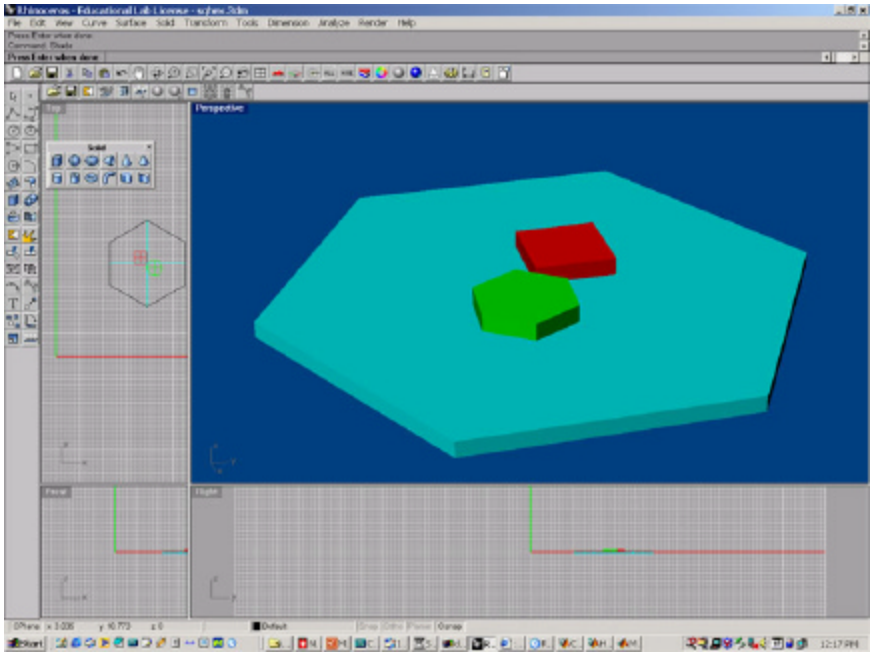

Figure 10: Geometry import direct from Rhino (3D CAD) via raw triangles into Surface Evolver to facilitate fast iteration in design time.

Relevant and useful reading.

I have included a long list of papers that I have found useful as a general introduction to the field of self assembly.

1. [001] Three-Dimensional Self-Assembly of Complex, Millimeter-Scale Structures through Capillary Bonding. Oliver, S.R.J., Clark, T.D., Bowden,N., Whitesides, G.M., J.Am.Chem.Soc. 2001, 123, 8119-8120.
2. [002] Mesoscale Self-Assembly of Hexagonal Plates Using Lateral Capillary Forces: Synthesis Using the "Capillary Bond". Bowden, N., Choi, I.S., Grzybowski, B.A., Whitesides, G.M., J.Am.Chem.Soc., 1999, 121, 5373-5391
3. [003] Design of Three Dimensional, Millimeter-Scale Models for Molecular Folding. Clark, T.D., Boncheva, M., German, J.M., Weck, M., Whitesides, G.M., J.Am.Chem.Soc., Vol.124, No.1., 2002.,18-19.
4. [004] Molecule-Mimetic Chemistry and Mesoscale Self-Assembly. Bowden, N.B., Weck, M., Choi, I.S., Whitesides, G.M., Acc.Chem.Res., 2001, 34, 231-238.
5. [005] Fabrication of Micrometer-Scale, Patterned Polyhedra by Self-Assembly. Gracias, D.H., Kavthekar, V., Love, J.C., Paul, K.E., Whitesides, G.M., Adv.Mater. 2002, 14, No.3, Feb 5, 235-238
6. [006] Fabrication of Three-Dimensional Microstructures by Electrochemically Welding Structures Formed by Microcontact Printing on Planar and Curved Substrates., Jackman, R.J., Brittain, S.T., Whitesides, G.M., J.MEMS, Vol.7, No.2, June 1998. pp261.
7. [007] Microorigami: Fabrication of Small, Three-Dimensional, Metallic Structures. Brittain, S.T., Schueller, O.J.A., Wu, H., Whitesides, S., Whitesides, G.M., J.Phys.Chem.B 2001, 105, 347-350.
8. [008] Self-Assembly of an Operating Electrical Circuit Based on Shape Complimentarity and the Hydrophobic Effect. Terfort, A., Whitesides, G.M., Adv. Mater. 1998, 10, No.6.
9. [009] Crystallization of Millimeter-Scale Objects with Use of Capillary Forces. Tien, J., Breen, T.L., Whitesides, G.M., J.Am.Chem.Soc. 1998, 120, 12670-12671
10. [010] Assembly of Mesoscopic Analogues of Nucleic Acids. Weck, M., Choi, I.S., Jeon, N.L., Whitesides, G.M., J.Am.Chem.Soc. 2000, 122, 3546-3547
11. [011] Microfabrication through Electrostatic Self-Assembly. Tien, J., Terfort, A., Whitesides, G.M., Langmuir 1997, 13, 5349-5355
12. [012] Mesoscopic, Templated Self-Assembly at the Fluid-Fluid Interface, Choi, I.S., Weck, M., Xu, B., Jeon, N.L., Whitesides, G.M., Langmuir 2000, 16, 2997-2999
13. [013] Self-Assembly of Microscale Objects at a Liquid/Liquid Interface through Lateral Capillary Forces. Bowden, N., Arias, F., Deng, T., Whitesides, G.M., Langmuir 2001, 17, 1757-1765
14. [014] Self-Assembly of 10-micron-Sized Objects into Ordered Three Dimensional Arrays. Clark, T.D., Tien, J., Duffy, D.C., Paul, K.E., Whitesides, G.M., J.Am.Chem.Soc. 2001, 123, 7677-7682
15. [015] Design and Self-Assembly of Open, Regular, 3D Mesostructures, Breen, T.L., Tien, J., Oliver, S.R.J., Hadzic, T., Whitesides, G.M., Science, 7 May, 1999, Vol. 284, pp948

16. [016] Forming Electrical Networks in Three Dimensions by Self-Assembly, Gracias, D.H., Tien, J., Breen, T.L., Hsu, C., Whitesides, G.M. Science, 18 August 2000, Vol. 289, 1170-1172
17. The Chemical Basis of Morphogenesis, A.M.Turing, Phil.Trans.of the Royal Society of London, Series B, Biological Sciences, Vol.237, Issue 641 (Aug.14, 1952), 37-72.
18. Morphogenesis. Bard, J., Cambridge University Press, U.K., 1990.
19. Biomathematics: Mathematics of Biostructures and Biodynamics, Andersson et.al., Elsevier, 1999.
20. Cellular Automata Machines, Toffoli and Margolus, MIT Press, 1988.
21. Toward in vivo Digital Circuits Authors: Ron Weiss, George Homsy, Thomas F. Knight Presented: Dimacs Workshop on Evolution as Computation, Princeton, NJ, January 1999.
22. Wolfram, Stephen., "A New Kind of Science"., Wolfram Media. 2002.
23. Banks, Edwin. R., "Information processing and transmission in cellular automata" , PhD Thesis, Dept. Mechanical Engineering, MIT, 1971.
24. Theory of Self-Reproducing Automata. Von Neumann. University of Illinois Press, 1966
25. Jacobson, Homer. "On models of reproduction" 1958 American Scientist 46 255-284
26. Penrose L S, Penrose R "A self-reproducing analogue" 1957 Nature 179 1183
27. Penrose, L.S. "Self Reproducing Machines" Scientific American, vol.200, June 1959- pp 105-114
28. Microactuation by Continuous Electrowetting and Electrowetting: Theory, Fabrication, and Demonstration. Jung-Hoon Lee, PhD. Thesis. Mechanical Engineering, UCLA, 2000.

Appendice A. : Example .FE datafile.

Duohex4.fe

```
// duohex4.fe

// convert to hex bath
// add a contact angle to surface of floating hex

//same as duohex.fe except parts are farther apart and do not converge (cool)

// Circular, tilting, non-coaxial wetted pads. With gravity.
// Same as bga-10.fe, but with 2D lateral movement of upper pad
// and tilting as optimizing parameters.

// Upper pad represented with boundary.
// Liquid entirely bounded by facets.

evolver_version "2.11c"  // minimum Evolver version needed

// physical constants, in cgs units
parameter S_TENSION = 480    // liquid solder surface tension, erg/cm^2
parameter SOLDER_DENSITY = 1 // grams/cm^3
gravity_constant 980     // cm/sec^2

// configuration parameters
optimizing_parameter height = 0.02   // height of upper pad, cm
parameter radius = 0.10              // radius of pads, cm
optimizing_parameter x_offset = .25    // offset in x of upper pad
optimizing_parameter y_offset = .25    // offset in y of upper pad
optimizing_parameter tilt = 6  scale=10000    // tilt about x-axis, degrees
parameter depth = 0.1 //depth of fluid

// loads on upper pad, dynes
parameter pad_weight = 500  // weight of upper pad
parameter x_load = 0.0  // load in x direction on upper pad
parameter y_load = 0.0  // load in y direction on upper pad
parameter cg_z = .02    // height of center of gravity of upper
                    // chip above upper pad


// for the second part:
optimizing_parameter x2_offset = -.25    // offset in x of upper pad
optimizing_parameter y2_offset = -.25    // offset in y of upper pad
optimizing_parameter tilt2 = -3  scale=10000    // tilt about x-axis, degrees
parameter pad_weight2 = 500  // weight of upper pad
parameter x2_load = 0.0  // load in x direction on upper pad
parameter y2_load = 0.0  // load in y direction on upper pad
parameter cg_z2 = .02    // height of center of gravity of upper
                    // chip above upper pad



PARAMETER angle = 95   // interior angle between plane and surface, degrees
```

```
// upper pad energy
quantity pad_energy energy method vertex_scalar_integral
scalar_integrand: pad_weight*z - x_load*x - y_load*y

// lower pad
constraint 1
formula: z = 0

// upper pad
constraint 2
formula: sin(tilt*pi/180)*(y - y_offset) = cos(tilt*pi/180)*(z - height)

// rim of lower pad
constraint 3
formula: x^2 + y^2 = (10*radius)^2

// lower pad
constraint 4
formula: z = -depth

// upper pad2
constraint 5
formula: sin(tilt2*pi/180)*(y - y2_offset) = cos(tilt2*pi/180)*(z - height)

// upper pad rim
boundary 1 parameters 1
x1: x_offset + radius*cos(p1)
x2: y_offset + radius*sin(p1)*cos(tilt*pi/180)
x3: height + radius*sin(p1)*sin(tilt*pi/180)

// upper pad center point
boundary 2 parameter 1
x1: x_offset
x2: y_offset + p1*sin(tilt*pi/180)
x3: height + p1*cos(tilt*pi/180)

// 2nd upper pad rim
boundary 3 parameters 1
x1: x2_offset + radius*cos(p1)
x2: y2_offset + radius*sin(p1)*cos(tilt2*pi/180)
x3: height + radius*sin(p1)*sin(tilt2*pi/180)

// upper pad center point
boundary 4 parameter 1
x1: x2_offset
x2: y2_offset + p1*sin(tilt2*pi/180)
x3: height + p1*cos(tilt2*pi/180)

vertices
// liquid surface
1  10*radius*cos(0*pi/3) 10*radius*sin(0*pi/3) 0   constraints 1,3 fixed
2  10*radius*cos(1*pi/3) 10*radius*sin(1*pi/3) 0   constraints 1,3 fixed
3  10*radius*cos(2*pi/3) 10*radius*sin(2*pi/3) 0   constraints 1,3 fixed
4  10*radius*cos(3*pi/3) 10*radius*sin(3*pi/3) 0   constraints 1,3 fixed
5  10*radius*cos(4*pi/3) 10*radius*sin(4*pi/3) 0   constraints 1,3 fixed
```

```
6  10*radius*cos(5*pi/3) 10*radius*sin(5*pi/3) 0    constraints 1,3 fixed

// floaty bit
7  0*pi/3 boundary 1 fixed
8  1*pi/3 boundary 1 fixed
9  2*pi/3 boundary 1 fixed
10 3*pi/3 boundary 1 fixed
11 4*pi/3 boundary 1 fixed
12 5*pi/3 boundary 1 fixed
// center of upper pad
13   0    boundary 2 fixed
// center of mass of upper pad, raised up a bit from pad surface
14   cg_z  boundary 2 fixed bare pad_energy

// bottom of dish
101  10*radius*cos(0*pi/3) 10*radius*sin(0*pi/3) -depth   constraints 3,4 fixed
102  10*radius*cos(1*pi/3) 10*radius*sin(1*pi/3) -depth   constraints 3,4 fixed
103  10*radius*cos(2*pi/3) 10*radius*sin(2*pi/3) -depth   constraints 3,4 fixed
104  10*radius*cos(3*pi/3) 10*radius*sin(3*pi/3) -depth   constraints 3,4 fixed
105  10*radius*cos(4*pi/3) 10*radius*sin(4*pi/3) -depth   constraints 3,4 fixed
106  10*radius*cos(5*pi/3) 10*radius*sin(5*pi/3) -depth   constraints 3,4 fixed

// floaty bit2
207  0*pi/3 boundary 3 fixed
208  1*pi/3 boundary 3 fixed
209  2*pi/3 boundary 3 fixed
210 3*pi/3 boundary 3 fixed
211 4*pi/3 boundary 3 fixed
212 5*pi/3 boundary 3 fixed
// center of upper pad
213   0    boundary 4 fixed
// center of mass of upper pad, raised up a bit from pad surface
214   cg_z2  boundary 4 fixed bare pad_energy

edges  // defined by endpoints
// liquid surface
1    1  2  constraints 1,3   fixed no_refine
2    2  3  constraints 1,3   fixed no_refine
3    3  4  constraints 1,3   fixed no_refine
4    4  5  constraints 1,3   fixed no_refine
5    5  6  constraints 1,3   fixed no_refine
6    6  1  constraints 1,3   fixed no_refine

// upper pad edges
7    7  8  boundary 1 fixed no_refine
8    8  9  boundary 1 fixed no_refine
9    9  10 boundary 1 fixed no_refine
10   10 11 boundary 1 fixed no_refine
11   11 12 boundary 1 fixed no_refine
12   12 7  boundary 1 fixed no_refine

// surface edges
112   1  12
113   1  7
114   2  8
115   2  9
```

```
116   3  10
117   3  209
118   4  210
119   5  211
120   6  212
121   1  207
122   207 11
123   208 10


// upper pad radii
19   7 13 constraint 2 fixed no_refine
20   8 13 constraint 2 fixed no_refine
21   9 13 constraint 2 fixed no_refine
22  10 13 constraint 2 fixed no_refine
23  11 13 constraint 2 fixed no_refine
24  12 13 constraint 2 fixed no_refine

// 2nd upper pad edges
307    207  208  boundary 3 fixed no_refine
308    208  209  boundary 3 fixed no_refine
309    209  210 boundary 3 fixed no_refine
310   210 211 boundary 3 fixed no_refine
311   211 212 boundary 3 fixed no_refine
312   212 207  boundary 3 fixed no_refine

// upper pad2 radii
319   207 213 constraint 5 fixed no_refine
320   208 213 constraint 5 fixed no_refine
321   209 213 constraint 5 fixed no_refine
322  210 213 constraint 5 fixed no_refine
323  211 213 constraint 5 fixed no_refine
324  212 213 constraint 5 fixed no_refine

// dishbottom
101    101  102  constraints 3,4   fixed no_refine
102    102  103  constraints 3,4   fixed no_refine
103    103  104  constraints 3,4   fixed no_refine
104    104  105  constraints 3,4   fixed no_refine
105    105  106  constraints 3,4   fixed no_refine
106    106  101  constraints 3,4   fixed no_refine

//dish vertical edges
201 101 1 no_refine
202 102 2 no_refine
203 103 3 no_refine
204 104 4 no_refine
205 105 5 no_refine
206 106 6 no_refine

faces // defined by oriented edge loops to have outward normal
// lateral faces
1   1 114  -7 -113 tension S_TENSION
2   2 116  -9 -115 tension S_TENSION
3   3 118  -309 -117 tension S_TENSION
4   4 119 -310 -118 tension S_TENSION
```

```
5   5 120 -311 -119 tension S_TENSION
6   6 121 -312 -120 tension S_TENSION
20  115 -8 -114 tension S_TENSION
21  117 -308 123 -116 tension S_TENSION
22  112 -11 -122 -121 tension S_TENSION
23  -123 -307 122 -10 tension S_TENSION
24  113 -12 -112 tension S_TENSION

// upper pad
8   7 20 -19 fixed no_refine color green tension 0 constraint 2
9   8 21 -20 fixed no_refine color green tension 0 constraint 2
10  9 22 -21 fixed no_refine color green tension 0 constraint 2
11 10 23 -22 fixed no_refine color green tension 0 constraint 2
12 11 24 -23 fixed no_refine color green tension 0 constraint 2
13 12 19 -24 fixed no_refine color green tension 0 constraint 2

// upper pad2
208   307 320 -319 fixed no_refine color red tension 0 constraint 5
209   308 321 -320 fixed no_refine color red tension 0 constraint 5
210  309 322 -321 fixed no_refine color red tension 0 constraint 5
211 310 323 -322 fixed no_refine color red tension 0 constraint 5
212 311 324 -323 fixed no_refine color red tension 0 constraint 5
213 312 319 -324 fixed no_refine color red tension 0 constraint 5


// dish bottom
101 -106 -105 -104 -103 -102 -101  fixed no_refine color cyan tension 0

// dish sides
102 101 202 -1 -201 fixed no_refine color cyan tension 0
103 102 203 -2 -202 fixed no_refine color cyan tension 0
104 103 204 -3 -203 fixed no_refine color cyan tension 0
105 104 205 -4 -204 fixed no_refine color cyan tension 0
106 105 206 -5 -205 fixed no_refine color cyan tension 0
107 106 201 -6 -206 fixed no_refine color cyan tension 0

bodies // defined by oriented face list
1   101 102 103 104 105 106 107 1 2 3 4 5 6 8 9 10 11 12 13 20 21 22 23 24 208 209 210 211
212 213  volume pi*(10*radius)^2*depth  density SOLDER_DENSITY

read

hessian_normal

read "xyztorque.cmd"

re := refine edges where on_constraint 2

// Typical evolution
gogo := {r; u; u; w 0.01; r; u; g 150; w 0.005; r; u; g 20 }
```

Appendice B : Command file for datafile update.

Xyztorque.cmd

```
// Commands to calculate x, y, and z forces, and tilt torque.

// command to smoothly change x offset.
// Use: set new_x_offset to desired value, then do do_x_offset.
new_x_offset := x_offset;
do_x_offset := { doff := new_x_offset - x_offset;
                 x_offset := new_x_offset;
               slope := tan(tilt*pi/180);
                 set vertex x x + doff*z/(height + slope*(y - y_offset));
                 recalc;
             }
// Central difference, moving linearly
dx := 1e-5;
do_xforce := {
             new_x_offset := x_offset + dx; // do this first so constraint in right place
                 do_x_offset;
             new_energy_1 := total_energy - body[1].pressure *
                    (body[1].volume - body[1].target);
             new_x_offset := x_offset - 2*dx;
                 do_x_offset;
             new_energy_2 := total_energy - body[1].pressure *
                    (body[1].volume - body[1].target);
             xforce := -(new_energy_1 - new_energy_2)/2/dx;
             printf "xforce: %18.15f  (central difference, linear move)\n", xforce;
             new_x_offset := x_offset + dx; // restore original
                 do_x_offset;
           }

// command to smoothly change y offset.
// Use: set new_y_offset to desired value, then do do_y_offset.
new_y_offset := y_offset;
do_y_offset := { doff := new_y_offset - y_offset;
                 y_offset := new_y_offset;
                 set vertex y y + doff*z/(height + slope*(y - y_offset));
                 recalc;
             }
// Central difference, moving linearly
dy := 1e-5;
do_yforce := {
             new_y_offset := y_offset + dy; // do this first so constraint in right place
                 do_y_offset;
             new_energy_1 := total_energy - body[1].pressure *
                    (body[1].volume - body[1].target);
             new_y_offset := y_offset - 2*dy;
                 do_y_offset;
             new_energy_2 := total_energy - body[1].pressure *
                    (body[1].volume - body[1].target);
             yforce := -(new_energy_1 - new_energy_2)/2/dy;
             printf "yforce: %18.15f  (central difference, linear move)\n", yforce;
             new_y_offset := y_offset + dy; // restore original
                 do_y_offset;
```

```
        }

// Central difference, moving linearly
dz := 1e-5;
do_zforce := {
            set vertex z z + dz*z/height; // do this before changing height
          height := height + dz;
          recalc;
          new_energy_1 := total_energy - body[1].pressure *
                (body[1].volume - body[1].target);
            set vertex z z - 2*dz*z/(height + slope*(y - y_offset));
          height := height - 2*dz;
          recalc;
          new_energy_2 := total_energy - body[1].pressure *
                (body[1].volume - body[1].target);
          zforce := -(new_energy_1 - new_energy_2)/2/dz;
          printf "zforce: %18.15f  (central difference, linear move)\n",
                zforce;
            set vertex z z + dz*z/(height + slope*(y - y_offset));
          height := height + dz; // restore original
          recalc;
        }

// Torque
// First, a smooth tilting
new_tilt := tilt;
do_tilt := { dtilt := (new_tilt - tilt)*pi/180;
          slope := tan(tilt);
            tilt := new_tilt;
          foreach vertex vv do {
                beta := z/(height + slope*(y - y_offset));
                vv.y := y_offset + cos(beta*dtilt)*(y - y_offset)
                        - sin(beta*dtilt)*(z - beta*height);
                vv.z := beta*height + sin(beta*dtilt)*(y - y_offset)
                        + cos(beta*dtilt)*(z - beta*height);
                }
        }

// Torque by central difference, moving linearly
dangle := 1e-4; // degrees
do_torque := {
          new_tilt := tilt + dangle; // do this first so constraint in right place
            do_tilt;
          new_energy_1 := total_energy - body[1].pressure *
                (body[1].volume - body[1].target);
          new_tilt := tilt - 2*dangle;
            do_tilt;
          new_energy_2 := total_energy - body[1].pressure *
                (body[1].volume - body[1].target);
          torque := -(new_energy_1 - new_energy_2)/2/(dangle*pi/180);
          printf "torque: %18.15f  (central difference, linear move)\n", torque;
          new_tilt := tilt + dangle; // restore original
            do_tilt;
        }

forces := { do_xforce; do_yforce; do_zforce; do_torque; }
```

Appendix C: MATLAB code for *.raw CAD import.

```
clear all;

fid = fopen('import.fe','w')
fprintf(fid,'vertices\n')
fclose (fid);

fid = fopen ('hexsq.raw');
a=fscanf(fid,'%g %g %g',[3 inf]);
fclose (fid);
a=a';
[row,col]=size(a);
b=[1:row];
b=b';
c=[b a];
c=c;
fid = fopen('import.fe','a')
fprintf(fid,'%i %8.6f %8.6f %8.6f fixed\n',c')
fclose (fid);

x=b;
for i=1:row;
    x(i,2)=b(i);
end

for i=0:row/3-1;
x(i*3 + 1,3) = b(i*3 + 2);
x(i*3 + 2,3) = b(i*3 + 3);
x(i*3 + 3,3) = b(i*3 + 1);
end

fid = fopen('import.fe','a');
fprintf(fid,'edges\n');
fprintf(fid,'%i %i %i fixed no_refine\n',x');
fclose(fid);

for i=1:row/3;
z(i,1)=i;
z(i,2)=3*i-2;
z(i,3)=3*i-1;
z(i,4)=3*i;
end

fid = fopen('import.fe','a');
fprintf(fid,'faces\n');
fprintf(fid,'%i %i %i %i color 4 backcolor 6 fixed no_refine\n',z');
fclose(fid);
```