# Hypertoons in Python

**By Kirby Urner**

**July 25, 2005**

Hypertoons represent the convergence of several themes for me, one of them being collaborating on open source projects in Python.

Back in March 1999, I published an article in *FoxPro Advisor*, which stood out, then as now, for its departure from the norm.[1] I was using Microsoft FoxPro to write scene description language for POV-Ray, the well-known ray tracing and rendering program.[2] That in itself was unusual, but on top of that, I was exploring a newfangled coordinate system my friends and I had dubbed the quadray coordinate system, or quadrays for short.[3] A core theme of this *Advisor* article was how such mathematical content became more lucidly accessible and interactive to students given a shell-based programming language. A few months later, I discovered Python.

Earlier, in 1996, I was trying to win a contest for a free SGI workstation, with encouragement from my friend Richard Hawkins, an artist and geometer.[4] The idea I used to try and win the workstation (I didn't win) was that of the hypertoon.[5] A hypertoon consists of key frames and transformations between them, randomly played, with the key frames serving as switch points. An analogy: jump on a subway train to any station; when you arrive at your destination, randomly choose any train departing from that station, and so on (it's OK to visit the same stations multiple times). Even back in 1996, my focus was synergetic geometry, R. Buckminster Fuller's philosophical transformer cartoons. My coding skills weren't up to creating a demo however (that's what I thought the SGI workstation would help me create).

Finally, almost 10 years later, I was able to code a demo hypertoon, using Python 2.4, and an experimental version of VPython.[6] In the intervening years, I'd switched the development of quadray coordinates to a set of Python classes, and continued to work with POV-Ray as my principal engine for graphical output. At the peak of my technique, I was able to provide Design Science Toys, a company run by my friend Stuart Quimby, with attractive renderings of StrangeAttractors, a magnetic geometry toy just coming to market. This work with POV-Ray meant I had a large assortment of polyhedra already built up as classes, on top of my quadray classes. By repurposing and merging these modules, I was able to accommodate the simple, friendly VPython API in less than a day.

 

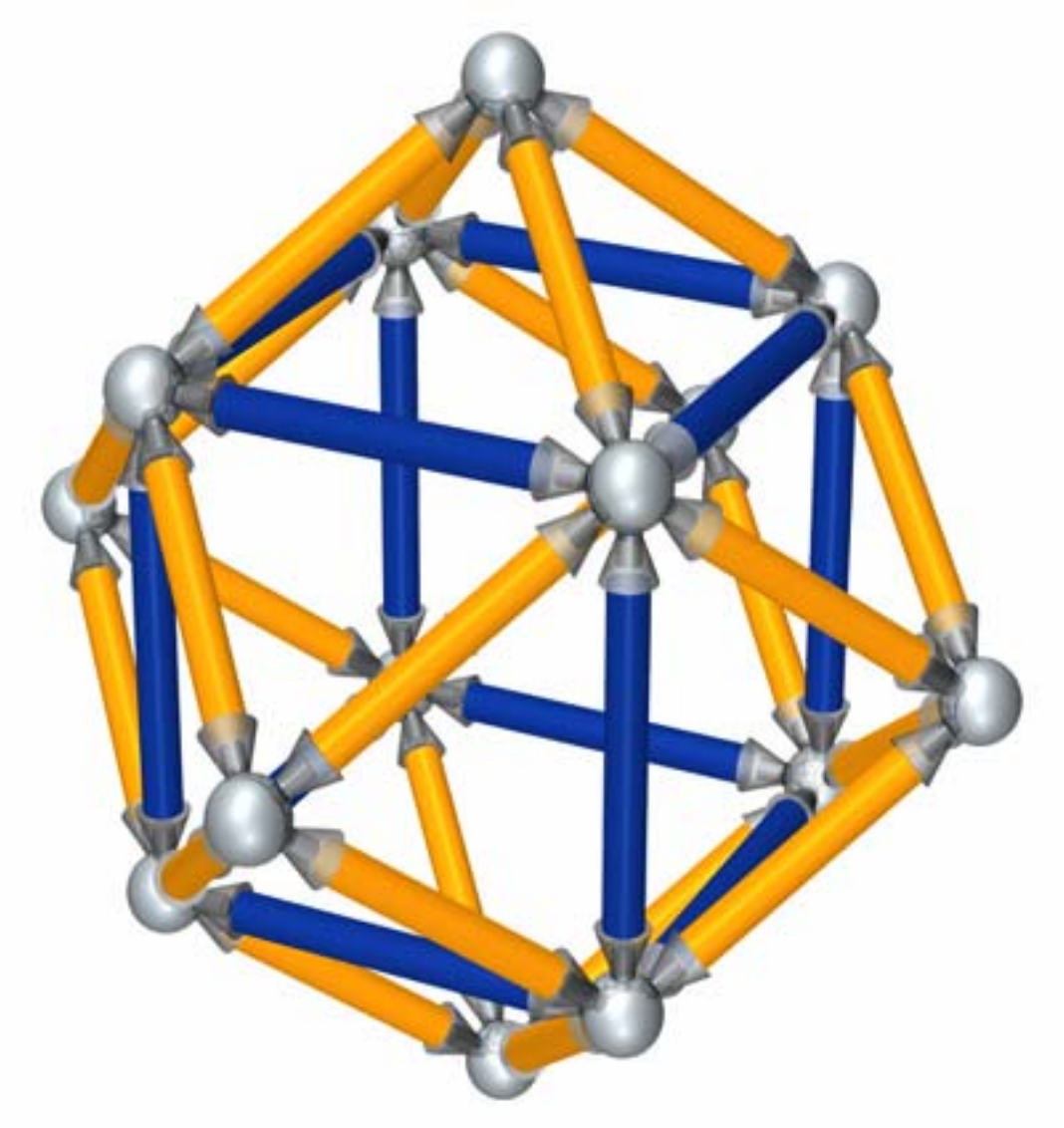

Fig 1: *Python + POV-Ray: artwork for a toy*

VPython, in case you don't know, is a C++ library with Python bindings, that does a lot of what OpenGL does, but with a very streamlined and simplistic API.[7] The VPython web site slogan is "3D Programming for Ordinary Mortals." If all you need are cylinders, spheres, rectangular prisms, a few other objects, and minimal control of lighting and scene, then VPython may be the handiest tool you've run across. I found this was so for me. Developed at Carnegie Mellon, VPython is used in physics courses by non-CS majors. A VPython scene may be interactively explored, using zoom in/out and rotate controls, even while Python code is changing what you're seeing. In this sense, it's a lot like VRML, only more dynamic, and without the complexity.

Let me zoom out and provide some more context. I used to be a high school math teacher, pre the open source revolution, and I still consider myself an educator and occasionally get teaching jobs. I think a lot about how computer languages, especially those with an interactive shell, might be used to boost the math curriculum. *Mathematica*

has already had an accelerating effect at higher levels. But why not use a general purpose, free language like Python? As an active participant on edu-sig (a Python community mail list), and maintainer of the edu-sig home page, I think about these issues a lot.[8]

I should also explain something about Bucky Fuller's synergetic geometry. When I go into a math class, perhaps at the elementary school level, the crux of my presentation has consisted of stiff, paper-board polyhedra, hinged with glue and electrical tape, with one side left open. These containers are precisely proportioned, such that they tend to have simple, whole number relationships to one another, volume-wise. The tetrahedron fills the cube with exactly three pours – I use dried beans as my substance. The cube fills the rhombic dodecahedron in two pours (I pour beans from one polyhedron to another). The octahedron and cuboctahedron have volumes of 4 and 20 respectively, relative to the tetrahedron's, which is pegged at unity.[9]

How do quadrays fit into this picture? In the familiar XYZ system, we shoot basis vectors from the origin to the six corners of the regular octahedron. The three positive rays are labeled (1,0,0), (0,1,0) and (0,0,1) respectively, with corresponding negative counterparts shooting off at 180 degrees to these three. In the quadray system, we shoot four vectors from the origin to the vertices of the regular tetrahedron, and label these points (1,0,0,0) (0,1,0,0) (0,0,1,0) and (0,0,0,1). We don't need any negative numbers. Every point in space may be expressed as a linear combination of these four. One might argue that these four vectors are not themselves linearly independent, but that's not important. As a mechanism for representing points, and therefore polyhedra, this is a workable notation and provides a basis for a Python class, complete with operator overloading.

 

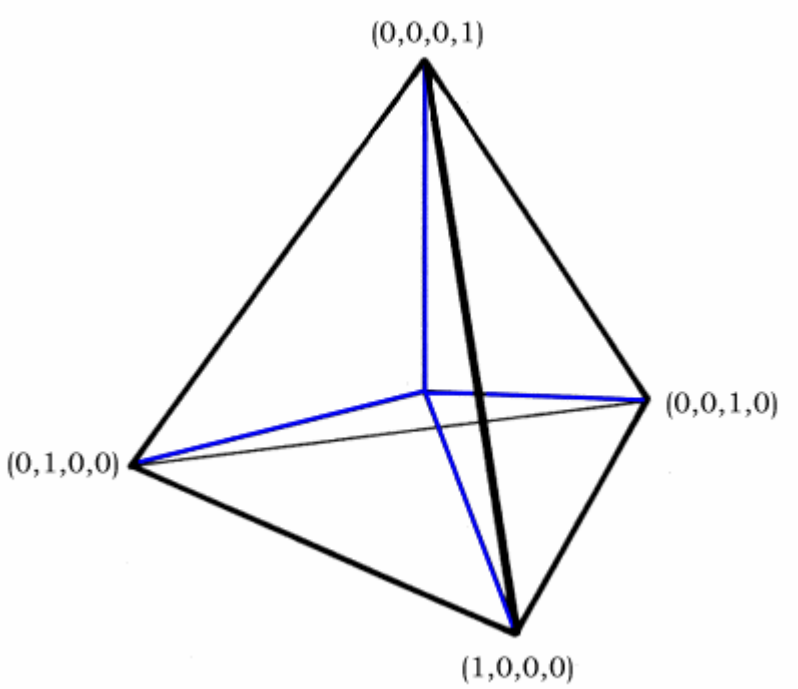


Fig 2: *Quadray Coordinate System: Used to Map Vertices of Polyhedron*

When I build the polyhedra of Fuller's concentric hierarchy, as the key frames of my VPython hypertoon, I use quadray coordinates to start. I have 26 primary vertices, labeled A-Z, plus a host of subsidiary vertices computed with A-Z as a starting point.[10] These all get saved in a global dictionary, and polyhedra get defined by their faces, using letters to go around each face. Shown below, is Python code defining the initial points A-Z, followed by a Tetrahedron class as defined in rbf.py (rbf = R. Buckminster Fuller):

```
def init():
    global Vset

    ORIGIN = vector(0,0,0)

    A = vector(Qvector((1,0,0,0)).xyz)  # origin to corner of tetra
    B = vector(Qvector((0,1,0,0)).xyz)  #          "
    C = vector(Qvector((0,0,1,0)).xyz)  #          "
    D = vector(Qvector((0,0,0,1)).xyz)  #          "

    # tetrahedron's dual (also a tetrahedron i.e. inverted tet)
    E,F,G,H = B+C+D,A+C+D,A+B+D,A+B+C

    # tetrahedron + dual (inverted tet) = duo-tet cube

    # octahedron vertices from pairs of tetrahedron radials
    I,J,K,L,M,N = A+B, A+C, A+D, B+C, B+D, C+D

    # octahedron + dual (cube) = rhombic dodecahedron

    # cuboctahedron vertices from pairs of octahedron radials
    O,P,Q,R,S,T = I+J, I+K, I+L, I+M, N+J, N+K
    U,V,W,X,Y,Z = N+L, N+M, J+L, L+M, M+K, K+J
```

```
class Tetra(Shape):
    """
                            Labels of   Numbers of
    Shape          Volume   Vertices    Vertices, Edges, Faces
    ------------------------------------------------------------
    Tetrahedron        1      A-D           4       6      4
    """

    faces = [('A','B','C'),('A','C','D'),('A','D','B'),('B','C','D')]

    sphcolor = cylcolor = "Orange"

    def __init__(self):
        Shape.__init__(self)
        self.volume   = 1.0
```

The hypertoon module imports rbf.py as a stash of polyhedra, already proportioned and oriented per the synergetics concentric hierarchy. All that remains is to define the key frames and the transitions between them. I developed some 30 transitions between about 8 key frames.

A key frame is some polyhedron, say an icosahedron. A transition takes us from one polyhedron to another (analogy: a subway between key frame stations). For example, from the icosahedron you might inscribe its dual, the pentagonal dodecahedron, or you might collapse it to an octahedron by way of what Fuller called the jitterbug transformation, or you might expand it into a cuboctahedron, also by way of the jitterbug transformation.[11] That's three possible transitions, so a randomizer will need to choose one of them. Next time through this same key frame, the choice might be different.

From the viewer's point of view, the motion is smooth, with no hesitations suggesting decision points in key frames. The infrastructure is all below the surface.

Of course setting up a hypertoon requires a Python data structure, a dictionary of dictionaries in my implementation. Every time the play head encounters a key frame, it needs to know which transitions to choose from. In other words, each named key frame is paired with a value: the dictionary of valid, named transitions to other key frames.

In my first draft, I used a lengthy chunk of code to populate the global dictionary. However, Scott David Daniels, a participant on edu-sig, proposed and coded an enhancement using decorator syntax (new as of Python 2.4) to automatically run each transition through a function that would auto-populate the hypertoon dictionary.

The function returned by the decorator function is itself a function, which accepts the transition function as input and returns it intact, so it still runs per usual. However, in the meantime, the first invocation passes the names of the two key frames connected by the transition, i.e. where the transition starts and ends. This information populates the dictionary used by the hypertoon function at runtime.

Below is one example transition, taking us from a rhombic dodecahedron (‘rh’) to an octahedron (‘octa’), inscribed as long diagonals. The rhombic dodeca first draws itself, then deletes any shape in the scene that preceded this key frame (i.e. an earlier rhombic dodecahedron – the endpoint of the previous transition). Then it traces an octahedron by lengthening VPython cylinders, giving an Etcha-Sketch style animation. Finally, the rhombic dodecahedron is deleted, leaving only the octahedron. This octahedron object will be returned, and used as input to the next transition, which will start by drawing a new octahedron and deleting this one (such overlap is necessary to maintain visual continuity through key frames).

```
@transition('rh', 'octa')
def trans0(thing):
    """trans0: rh -> octa"""
    rh = rbf.Rhdodeca()
    # A: rh dodeca all alone
    rh.draw()
    if thing is not None:
        thing.delete()
    oc = rbf.Octa()
    oc.draw(trace=True)
    rh.delete()
    # B: octahedron all alone
    return oc
```

Note the transition function, invoked with decorator syntax, with the start and end key frames as arguments. The code below uses these arguments to populate the transitions dictionary, which is used at runtime to randomly choose segments as the hypertoon plays.

```
transitions = {}
verbose = False

def transition(initial, final):
    def record_transition(transition):
        try:
            surprise = transitions[initial][final]
        except KeyError:
            pass # this is expected
        else:
            print >>sys.stderr, 'Changing %s -> %s: from %r to %r' % (
                initial, final, surprise.__name__, transition.__name__)
        transitions.setdefault(initial, {})[final] = transition
        return transition
    return record_transition
```

What’s happening here? A function is manufactured and returned, named *record_transition*, the job of which is to accept and return a transition (e.g. trans0 above). The manufactured function is customized, per arguments, to populate a dictionary of dictionaries, named *transitions*, using a specific initial named frame as the primary key, and the final frame as a secondary key (the same initial frame pairs with multiple final

frames). The value associated with the final frame is the transition itself i.e. is a pointer to the actual function object in memory.

This implementation expects a unique transition along any edge between nodes and flags a problem to stderr if a second transition connecting the same two dots comes down the pike. Treating multiple edges between the same two nodes as an error is not intrinsic to the generic hypertoon idea (one may imagine getting from a cube to a rhombic dodecahedron by more than one route), however I find this an acceptable simplification, the payoff being we're allowed to treat (initial, final) tuples as the unique names of transitions. There's no "one right way" to implement the hypertoon concept.

The hypertoon itself may be run as a thread, and depending on a command line argument, I've optionally implemented a two-thread version, i.e. two versions of the hypertoon run simultaneously, juxtaposed in the same scene. If your computer is powerful enough, more threads might be added, although this inevitably introduces lags and less fluid motion – a tradeoff. VPython optionally supports a couple of stereo modes, including red-blue, viewable with inexpensive cardboard glasses.[12] Both full screen and window modes are available.

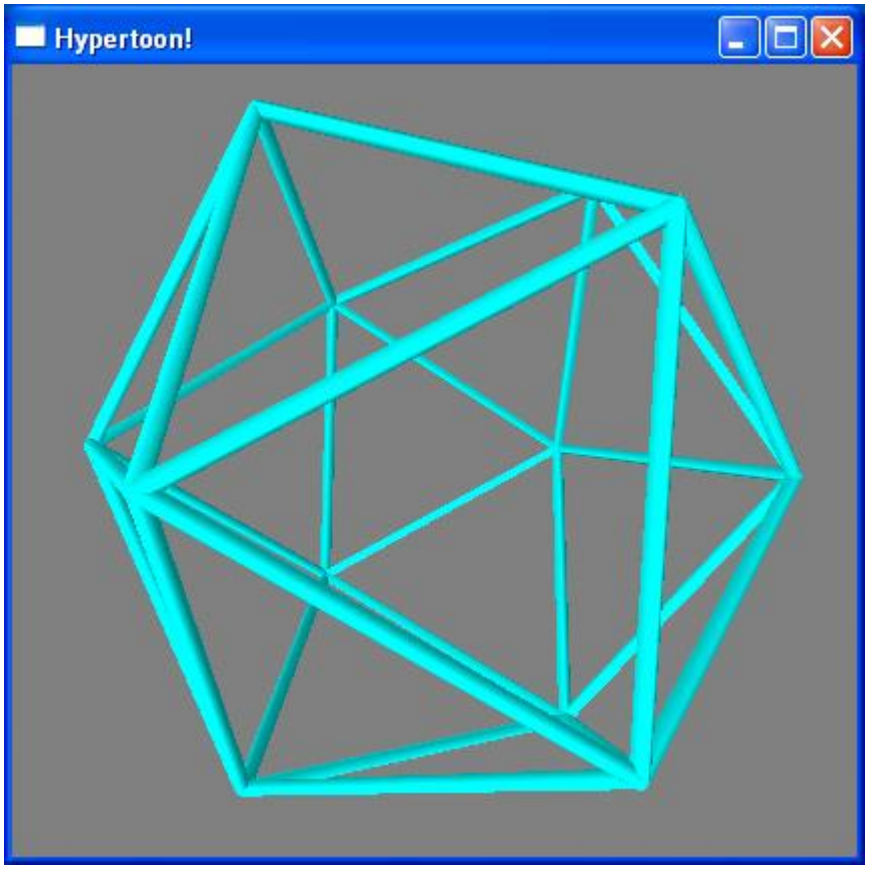

Fig 3: *Key frame: Icosahedron*

 

Here's the core of the hypertoon, the hypertoon function, also reworked by Scott to leverage the decorator-driven transitions data structure, already fully populated at runtime:

```
def hypertoon(repetitions, node=None):
    """
    Start an N-transition hypertoon.  node may be a name
    like 'cube'
    """
    liveobject = None
    if node is None:
        node = choice(transitions.keys())
    for x in range(repetitions):
        possibilities = transitions[node]
        coming, function = choice(possibilities.items())
        if verbose:
            print node, coming, function.__name__, function.__doc__
        liveobject = function(liveobject)
        node = coming
```

The hypertoon loops for n repetitions, starting at a pre-specified node, or at some random one, and choosing from among the possible transitions on file for that node. The variable *coming* names the key frame we'll be arriving at, once the randomly chosen transition is finished (note that *choice* is imported from the standard *random* module). This destination node then becomes the new starting point (last line: *node = coming*), and so on, as the hypertoon function randomly iterates through our data structure, and as VPython renders the corresponding geometry cartoons to the scene window or to the full screen (the VPython API lets you specify either target).

The *liveobject* variable refers to the last polyhedron drawn in any given transition. This object gets passed to the next transition as parameter 'thing' so that it might be deleted, but usually only after it has first been redrawn, making this first deletion undetectable by the viewer. This partial overlap in content between transitions is what keeps the motion smooth. The scene is never empty, even when moving from one transition to the next.

Drawings and deletions are accomplished within the rbf module.[13] A polyhedron will consist of several VPython objects, all of which have to be recorded, so they may be deleted explicitly when no longer needed. I use class/object syntax to manage my polyhedra, with *draw*, *delete* and *_trace* methods. I invoke _trace from within a transition when I want the edges of a polyhedron to appear gradually, by a process of cylinder elongation. This requires some control over frame rate, plus a stepped change in the VPython's vector.axis parameter:

```
    def _trace(self, a, b, r, c, therate=500):
        v0 = self.vertices[a]
        v1 = self.vertices[b]
        start = cylinder(pos=v0, length=0, color=c, radius = r)
        self.garbage.append(start)
        self.cyls[(a,b)] = start
        v3 = v1-v0
        for i in range(1,101):
```

```
        rate(therate)
        start.axis = v3*i/100.0
```

The vector mathematics employed in rbf.py is fairly straightforward. I start out at the top with quadrays, but by runtime I'm working with VPython's vector objects, which are of course x,y,z-driven. The Vpython vector class takes a little getting used to, in that a cylinder is not defined by start and finish points (as is the case in POV-Ray), but by a start point paired with an axis, which supplies length and direction. The axis is basically the cylinder you want to end up with, anchored at the origin, with the starting point serving as an offset.

In my experience, showing Fuller's concentric hierarchy of polyhedra in the form of a hypertoon helps newcomers appreciate the nuances of his way of arranging and relating polyhedra. Thanks to the open source revolution and a culture of collaboration, the software keeps getting more powerful, such that an idea like Hypertoons is able to reach demo stage without a lot of overhead in terms of budget or manpower. I found that VPython's super simple API made this initial working demo relatively easy to develop, in part by repurposing older code already built around polyhedra, but outputting through a different API (POV-Ray's scene description language).

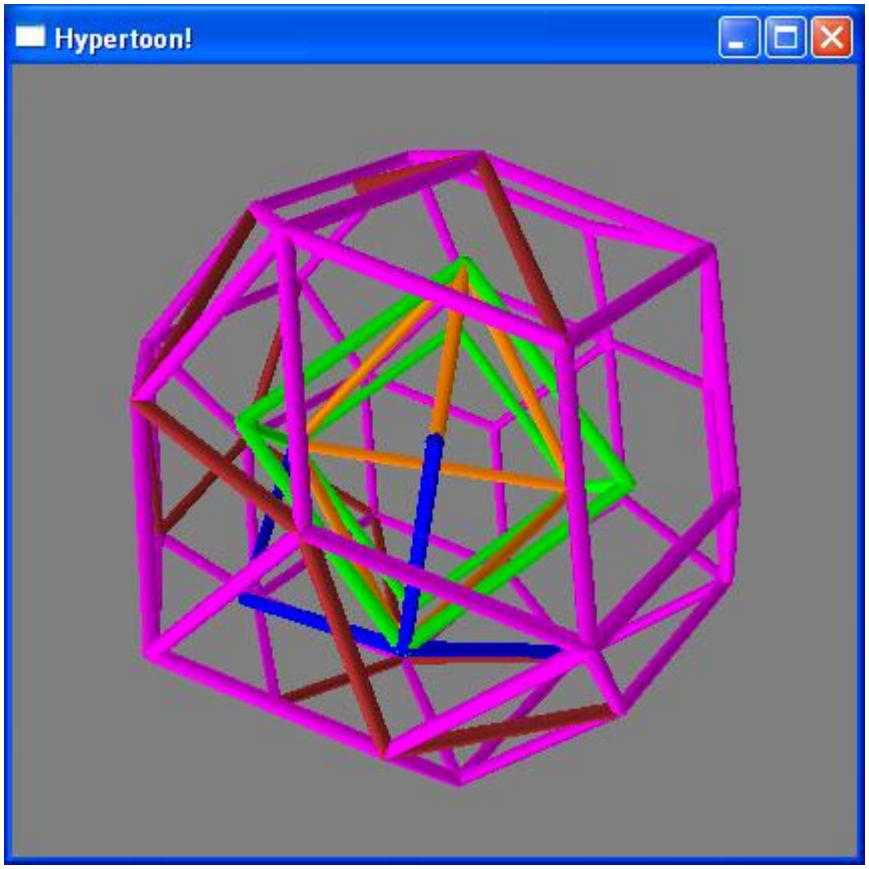


*Fig 4: Two threads in action: Rhombic dodecahedron partially drawn in blue, pentagonal dodecahedron drawing itself in dull red*

This entire development cycle reinforces my belief that open source, and Python in particular, have become invaluable and indispensable tools for math teachers and curriculum writers. Our ability to communicate concepts and techniques is greatly amplified by interactive, general purpose languages, augmented by specialized libraries, and especially libraries aimed at providing graphical output, like VPython.

---

[1] *Teaching Object-Oriented Programming with Visual FoxPro: Math mixes with OOP in this unusual use of Visual FoxPro* by Kirby Urner
http://advisor.com/Articles.nsf/ID/FA9903.URNEK01

[2] POV-Ray home page: http://www.povray.org/

[3] *An Introduction to Quadrays*
http://www.grunch.net/synergetics/quadintro.html

[4] Richard's many geometry animations are stashed here:
http://www.newciv.org/Synergetic_Geometry/ plus he's contributed many graphics to my own website http://www.grunch.net/synergetics/

[5] *On the Concept of HyperToon* by Kirby Urner, December 7, 1996
http://www.grunch.net/synergetics/hypertoon.html

[6] The two modules needed to run the demo, in addition to VPython itself, are available here:
http://www.4dsolutions.net/ocn/python/hypertoons/

[7] VPython home page: http://www.vpython.org/

[8] http://www.python.org/sigs/edu-sig/ is the Edu-sig home page, with links to the edu-sig mail list

[9] For more on these volume relationships, see my http://www.grunch.net/synergetics/volumes.html

[10] These 26 points are show in a graphic on this page: http://www.4dsolutions.net/ocn/oop7.html

[11] Fuller's magnum opus is on-line. Here's a graphic from that work, showing the jitterbug transformation:
http://www.rwgrayprojects.com/synergetics/s04/figs/f6008.html

[12] Dr. John Zelle was instrumental in getting stereo features coded into VPython. He also added code to make this hypertoon implementation more version-agnostic, with respect to Python and VPython. However, I came to the conclusion that VPython 2.4 experimental was doing a better job displaying the jitterbug transformation, which is why I do not mention Python 2.3 in the main body of this paper.

[13] rbf.py, a work in progress, is named for R. Buckminster Fuller