Ruby Quiz

Geodesic Dome Faces (#3)

by Gavin Kistner

SUMMARY

Given the faces for a tetrahedron, octahedron, or isocahedron, create a geodesic dome of arbitrary frequency.

The (equilateral triangle) faces of each primitive are given as triplets of vertex points. Each vertex is itself a triplet of cartesian 3-space coordinates, all of unit-distance from 0,0,0. (See the supplied points at the end for an example.) The resulting geodesic should be an array of triangular faces; again each face is a triplet of points, and each point is unit-distance from 0,0,0.

DETAILS

A 'simple' solution (using recursion) exists to subdivide each primary face into 4^n sub-faces. Instead, the following (more flexible) algorithm should be used, which allows for n^2 sub-faces:

[See http://phrogz.net/CSS/Geodesics/index.html for a visual example of the following algorithm.]

Step 1) Start with the three points defining a primary face.

Step 2) Divide each side of the face into equal length pieces; the number of pieces is specified by the 'frequency'. (A frequency of 0 subdivides the face not at all, a frequency of 1 divides each side into two equal pieces, a frequency of 2 into three equal pieces, and so on.)

Step 3) Connect each division point along two sides with a line that is parallel to the third side.

Step 4) Repeat with lines parallel to all three sides.

Step 5) New points are defined wherever the lines intersect.

The combination of the initial face points, the edge points, and the intersection of the connecting lines provide the points for the faces of the geodesic. (As diagrammed in http://phrogz.net/CSS/Geodesics/index.html#step5, the 16 faces created by subdividing the primary face with frequency 3 can be described as Aqm, qfm, qrf, rgf, rsg, szg, sBz, mfn, fhn, fgh, gyh, gzy, nho, hxo, hyx, oxC.)

All points should be 'normalized', so that they are unit-distance from the origin.

For extra points, ensure that the points for each face are always specified in the same direction, clockwise or counter-clockwise when looking from the origin. (The above list of faces are all specified in a clockwise direction.)

STARTER DATA

The points for the three primitives follow. (Solving for any one of them solves for all of them.)

```ruby
SQRT2 = Math.sqrt(2)
SQRT3 = Math.sqrt(3)
TETRA_Q = SQRT2 / 3
TETRA_R = 1.0 / 3
TETRA_S = SQRT2 / SQRT3
```

```
TETRA_T = 2 * SQRT2 / 3
GOLDEN_MEAN = (Math.sqrt(5)+1)/2

PRIMITIVES = {
    :tetrahedron => {
        :points => {
            'a' => Vector[ -TETRA_S, -TETRA_Q, -TETRA_R ],
            'b' => Vector[  TETRA_S, -TETRA_Q, -TETRA_R ],
            'c' => Vector[        0,  TETRA_T, -TETRA_R ],
            'd' => Vector[        0,        0,        1 ]
        },
        :faces => %w| acb abd adc dbc |
    },
    :octahedron => {
        :points => {
            'a' => Vector[  0,  0,  1 ],
            'b' => Vector[  1,  0,  0 ],
            'c' => Vector[  0, -1,  0 ],
            'd' => Vector[ -1,  0,  0 ],
            'e' => Vector[  0,  1,  0 ],
            'f' => Vector[  0,  0, -1 ]
        },
        :faces => %w| cba dca eda bea
                      def ebf bcf cdf |
    },
    :icosahedron => {
        :points => {
            'a' => Vector[  1,  GOLDEN_MEAN, 0 ],
            'b' => Vector[  1, -GOLDEN_MEAN, 0 ],
            'c' => Vector[ -1, -GOLDEN_MEAN, 0 ],
            'd' => Vector[ -1,  GOLDEN_MEAN, 0 ],
            'e' => Vector[  GOLDEN_MEAN, 0,  1 ],
            'f' => Vector[ -GOLDEN_MEAN, 0,  1 ],
            'g' => Vector[ -GOLDEN_MEAN, 0, -1 ],
            'h' => Vector[  GOLDEN_MEAN, 0, -1 ],
            'i' => Vector[ 0,  1,  GOLDEN_MEAN ],
            'j' => Vector[ 0,  1, -GOLDEN_MEAN ],
            'k' => Vector[ 0, -1, -GOLDEN_MEAN ],
            'l' => Vector[ 0, -1,  GOLDEN_MEAN ]
        },
        :faces => %w| iea iad idf ifl ile
                      eha ajd dgf fcl lbe
                      ebh ahj djg fgc lcb
                      khb kjh kgj kcg kbc |
    }
}
```

Quiz Summary

by Gavin Kistner

There were three major challenges which comprised the meat of this quiz:

1) What the heck was I talking about?

Although finding the answer to #1 was not intended to be part of the quiz, it certainly was. Due to incomplete specification and background, one had to do a bit of searching to understand what a geodesic dome is, and how that related to the supplied information about primitives and faces.

A quick search on Wikipedia yields:

Geodesic Dome

I won't repeat the information on that page here, but the summary is:

```
A geodesic dome is an approximation of a sphere, made out of triangles
connected by straight edges. To create a geodesic dome, start with a
platonic solid with triangular faces--tetrahedron, octahedron, or
icosahedron. Subdivide each triangular face into a number of smaller
triangles, and then push the points of those triangles out from the
center
of the primitive to the radius of the sphere.
```

This brings us to challenge number 2:

2) How do you find the points on the face, and make triangles from them?

There are various techniques for determining the exact locations of the points for the sub triangles. (Several techniques are described in the white paper "Geodesic Math", starting on page 9. The one chosen for and described by this quiz is the Class 1, Method 1 technique (page 9), which creates small triangles with sides of equal length.

Each of the four solutions was unique.

Everyone used vectors to solve the problem, although Dennis worked extra hard and created his own Vector class rather than using the one included in the matrix library included with Ruby.

Dennis also went his own way and divided the sides of the triangles into equal-angles (as measured from the center of the geodesic), instead of equal-length pieces. This technique is slightly more effective at evenly distributing the triangles across the surface of the sphere. For example, compare an octahedron subdivided with frequency 20, using the linear technique (as outlined by the quiz) versus the angular technique Dennis used in this picture. Note how the linear technique has the triangles piling up along the edges of the original face of the octahedron, where the radial technique does a better job of spacing them out.

I was the only one who used a recursive technique to hunt down all the triangles on the face, and was the only one to consider groups of faces as hexagons. The other three solutions all use cleaner methods of using one loop inside another and cleanly navigating the whole face, once.

Dennis and I both used a separate class for a face/triangle, while Warren and Martinus keep track of their information in a more 'raw' fashion.

3) How do you provide the results?

The solution by Martinus stands out, as he put the extra effort in to create a simple OpenGL viewer for his solution. While his solution wasn't quite correct (he forgot to normalize the points on the face, pushing them out away from the plane of the face to the surface of the sphere), it was a trivial change to fix it, as Dennis pointed out.

Further, Martinus' solution was the fastest (even with the additional code to normalize the points). In calculating the 20,808 faces for an octahedron of frequency 50, the times on my computer were:

```
Martinus:  4.32 seconds
   Gavin:  8.61 seconds
  Warren:  12.81 seconds
  Dennis:  80.49 seconds
```

For reasons I couldn't figure out, Dennis' solution returns a bunch of NaN points when dealing with the icosahedron primitive. (Which is why I used octahedrons to test the output.) Also, I had to add

```ruby
   attr_reader :faces
```

to his Dome class; without that (or an #instance_eval call) there was no way to get at the data, once created :)

It was difficult to test Warren's output, as the results it provided were not organized into triplets of points. The call to flatten took Warren's array of point-triplet arrays and squashed it into a flat array of points. However, without the call to flatten, the points were stuck in arrays of faces for each original face in the primitive. I had this problem initially with my solution, which is why I created a separate Face class -- so that flattening the arrays of arrays of faces would not squash out the array of points contained within each face.

An alternative solution, and I think what Warren intended, is something like Array#flatten_once, either through that separate library or by rolling ones' own #flatten_once method.

Despite the above 'problems', the quiz didn't specify what to do with the data in the end, or how to provide it. The solutions met the requirements correctly, if not conveniently :)

Wrap Up

When I proposed this quiz, I didn't realize how few people were conversant with the concepts of Geodesic domes (although I should have). I apologize for not being more clear with the specifications to the problem, particularly in the lack of background information.

I also did not see that the scope of the quizzes is intended to be as short as it is (30-60 minutes). Although this problem took some longer than I had expected, I *did* expect it to be a 2-3 hour problem. So, sorry for supplying a quiz which was perhaps more work than people have time for.

As penance, I give you some pretty pictures to end with: showing what the domes look like (using the linear interpolation defined by the quiz) for the three primitive types and with various frequencies. The first picture shows the faces clearly, the second shows how well they approximate spheres when used in a rendering engine that can smooth the shading between the faces.