The Problem of Calculating Vertex Normals for Unevenly Subdivided Smooth Surfaces

Ted Schundler
tschundler (a) gmail _ com

Abstract:
Simply averaging normals of the faces sharing a vertex does not produce ideal vertex normals. A smaller sized face should have less change in shading across it than a large one to produce a more evenly smoothed look. The nature of the problem is examined and solutions involving weighing the normals are explored.

Background:
To create the illusion of a smooth surface of a faceted object, the Phong method is what is most popularly used (even when other shaders are used to calculate the actual lighting given the normal). The normal vectors at the corner vertices of a 3D surface (triangle or quadrilateral) are interpolated to determine the normal at a given point. That interpolated normal is then used for shading calculation.

That works quite well, and needs no adjustment. The problem is choosing good vertex normals. Consider calculating the vertex normal for point B in the 2D example to the left (Figure 1). B is part of a set of line segments approximating part of a curve. If normals are calculated for segments AB and BD then the vertex normal at point B is the average of those normals. The normal varies by a uniform amount across the segments.

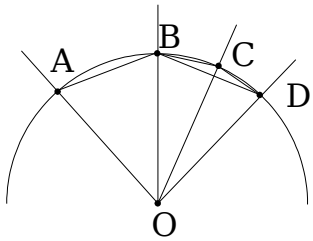


Figure 1

However, suppose the geometry is not even. Representing the same curve as A,B,D, now we use A,B,C,D. If we calculate segment normals for AB, BC, and CD. Then calculate vertex normals B and C by averaging them, there is a problem. Vertex normal C is correct, but B will instead point slightly towards A. Because of this, the change is angle over AB will be less than it should be, and the change in angle over BC will be more than it should be. If used for shading, it would produce uneven results.

For example, assume angles are relive to O, and A is at -40°, B is 0°, C is 20° and D is 40°. The segment normals are AB -> -20°, BC -> 10°, CD -> 30°, BD -> 20°. Using just segments AB and BD, B's normal is (-20 + 20)/2=0°. However with the new point C, B is (-20 + 10)/2=-5° (wrong), and C's normal is (10+30)/2=20° (correct).

The Issue & Approach:
The relative scale of the geometry needs to be taken into consideration when computing vertex normals. In the 2D case it is fairly straightforward. Normal of the smaller segment is going to me closer to that the normal at the vertex is. So, the shorter segment needs more weight. The multiplicative inverse of its length serves well as a weight for averaging the segment normals. Suppose AB has length 2, and BC has length 1. The weighted average for the values in the example above for B would be ((1/2)*-20 + (1/1)*10)/(1/2+1/1) = 0° (correct), and for C ((1/1)*10 + (1/1)*30)/(1/1+1/1) = 20° (correct).

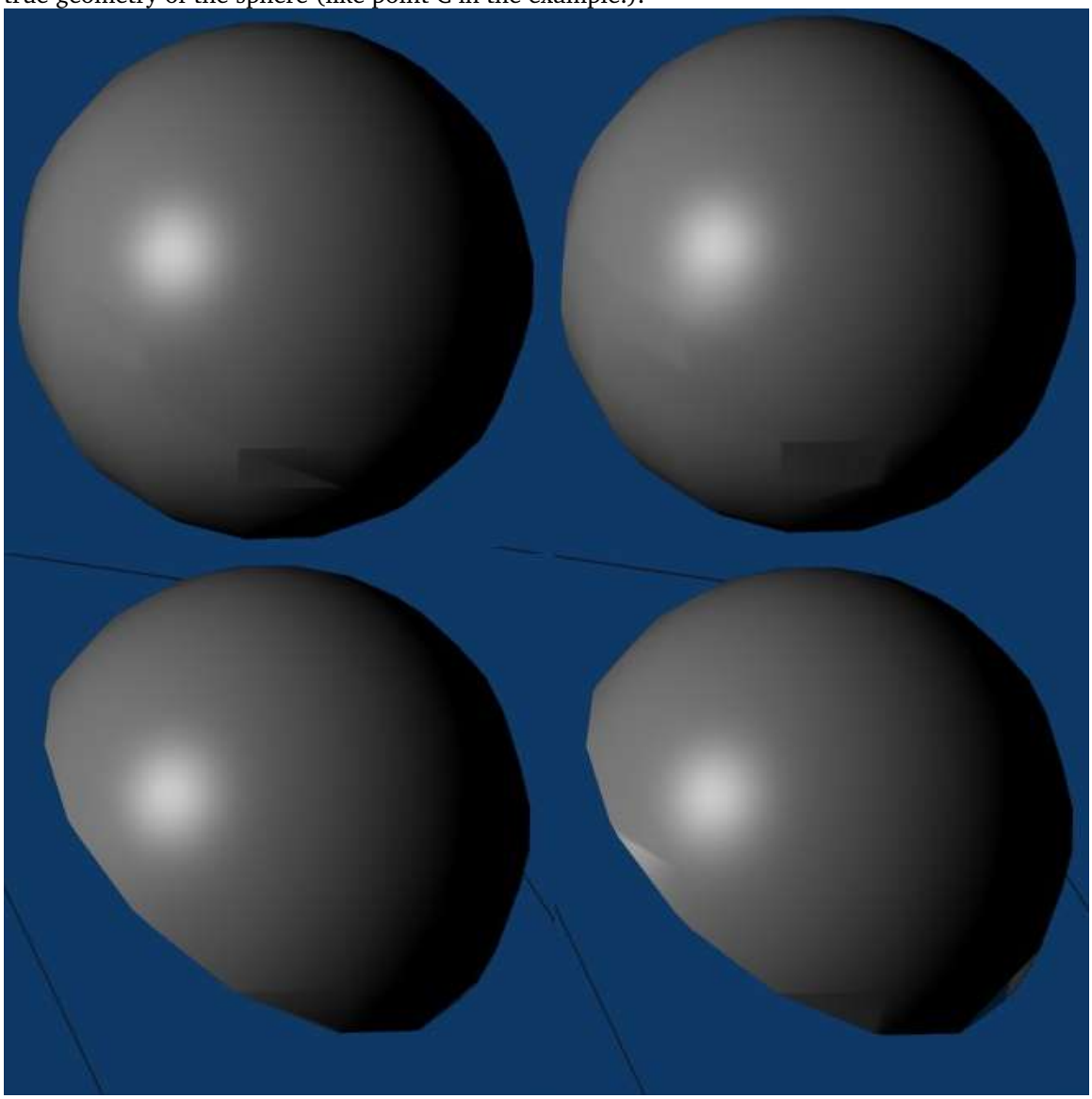
Unfortunately 3D is not so simple, and Google has failed to provide me with any better insight. At the moment, I don't know what the best solution is. I've tried out a number of approaches, each with their highlights and drawbacks. I'm presenting them hopefully to get some feedback from

others.
The basic premise of my approach is that smaller faces / edges need to have more weight. So, if you have a set of edges tugging at a vertex, the shortest edge should have the most pull.

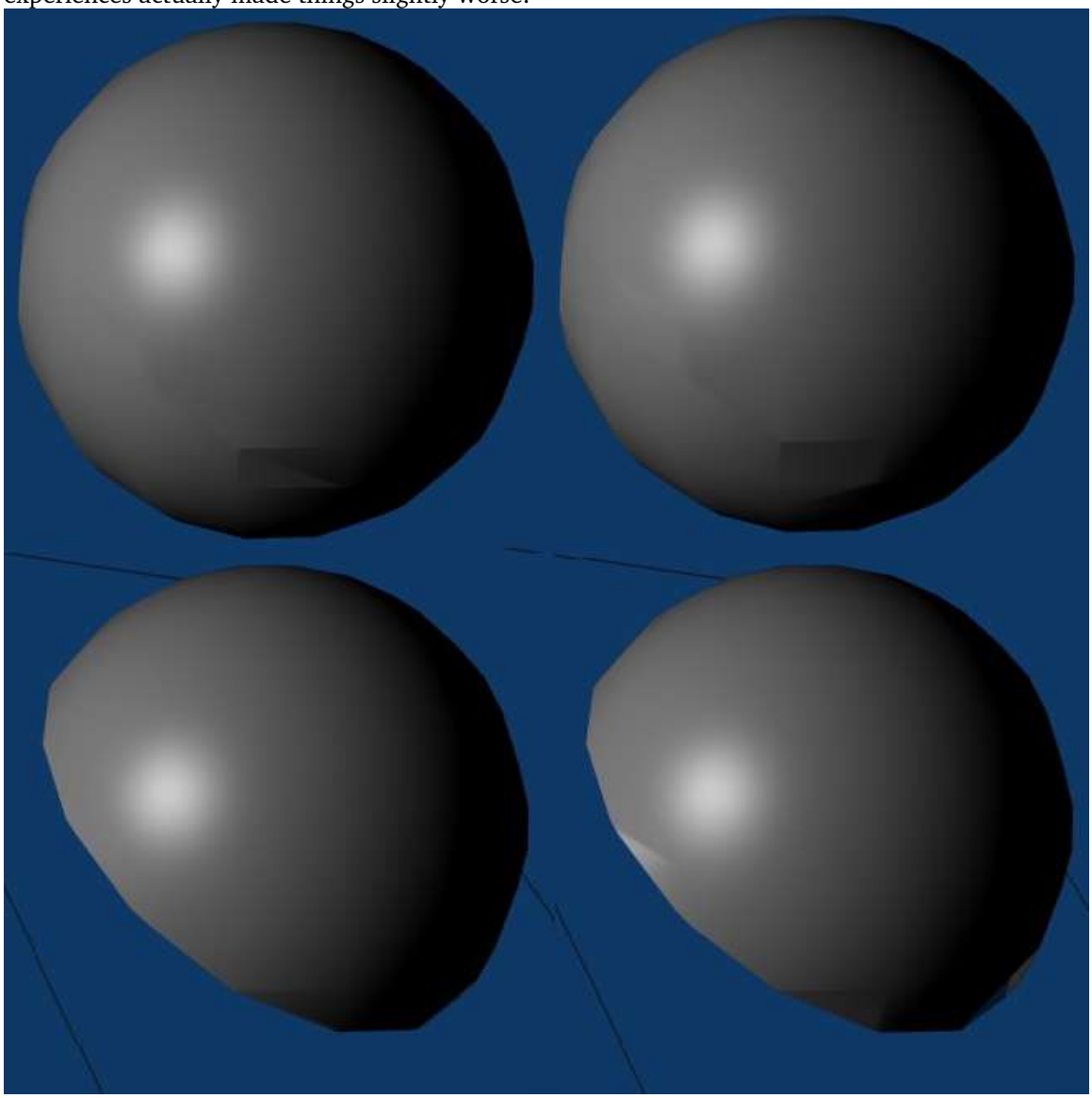
The Solutions:

Equal weights:

First, for comparison purposes, here are some test renders using equal weights on an unevenly subdivided sphere. The ones on the left are subdivided assuming flat faces. (how things are subdivided now if you use the knife tool for instance) The ones on the right are subdivided to the true geometry of the sphere (like point C in the example.).

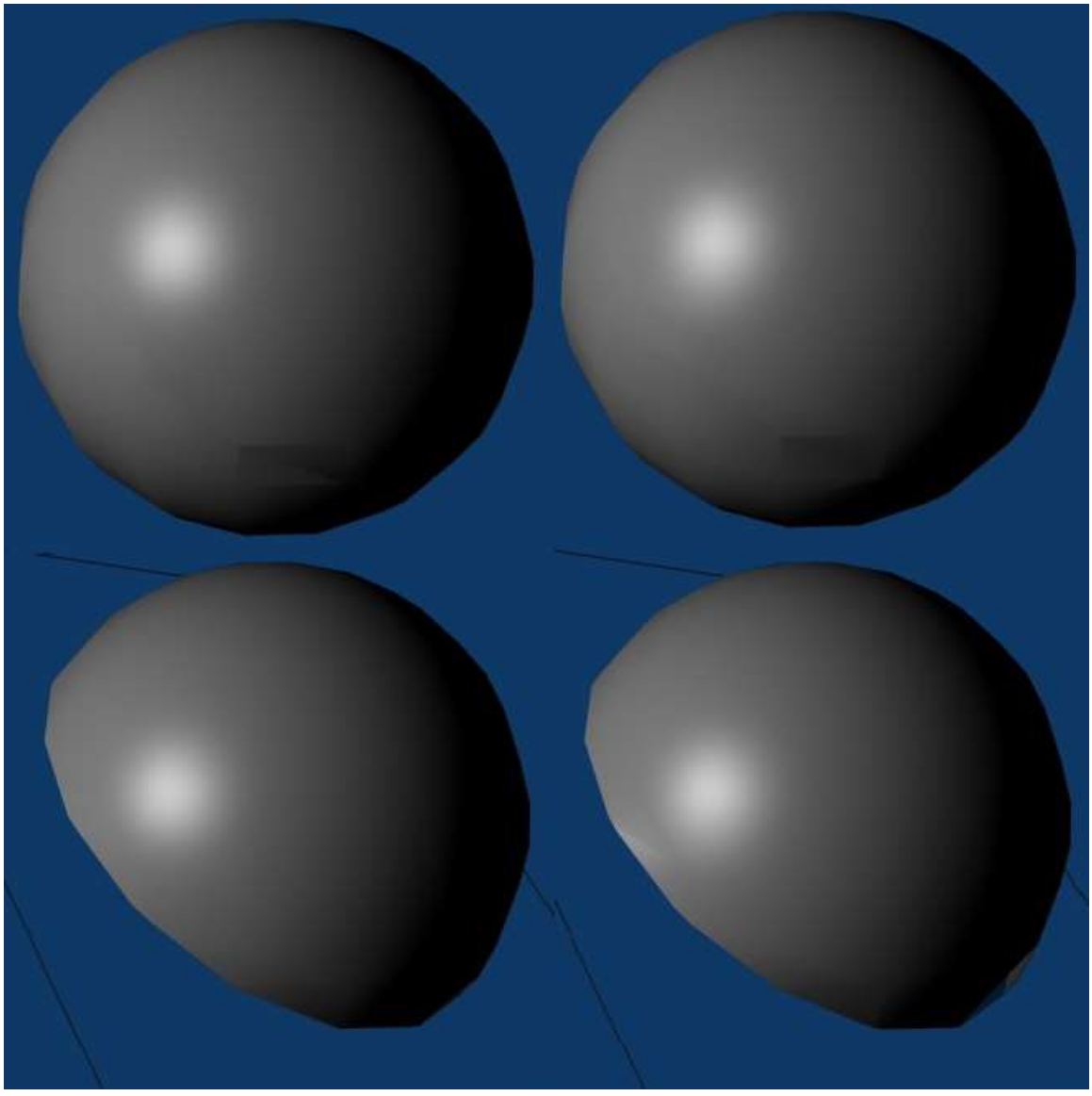


Angle between edges:
In Blender right now, convertBlenderScene.c has a normal weighting feature in normalenrender(), which seems to only kick in when AutoSmooth is on. (Otherwise it uses the averaging method that seems to be somewhere else in the code.) That code uses the $\cos^{-1}$ of the negative of the dot product. So, an angle of 0° has a weight of zero, and angle of 90° has a weight of π, and an angle of 180° has a weigh of 2π. I'm guessing the idea is that the normal between the edges that make up a wider span of the intersection should have a greater weight. It seems nice, but in my experiences actually made things slightly worse:



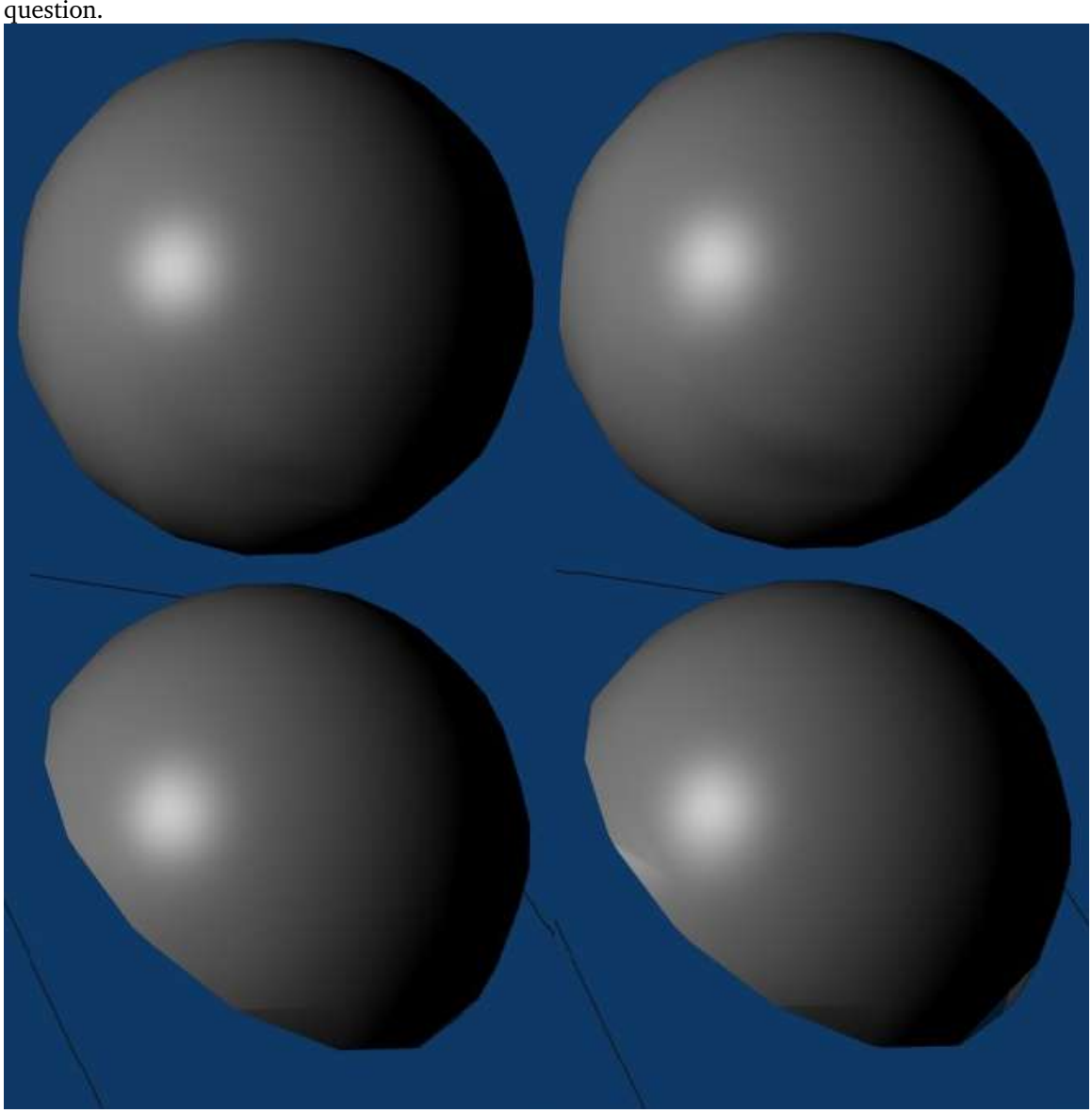
This method doesn't consider the size of the edges at all, and so in my case gives no improvement, and some virtually imperceptible extra distortion.. And I'm not sure that the reasoning is completely valid, but I suspect in general will produce good results. The reason being that if the angle is wide, the corresponding triangle will be flattened out more, and thus smaller. Because it's smaller, the amount of change across it should be less. So, in many cases it gives more weight where more weight is needed. The logic for giving it more weight however is slightly askew.

Sum of Inverse Lengths:
My first attempt was based on my 2D experiment with edge lengths. To stick with edges, the edge normal is the average of the normals of the adjacent faces (cross products with the two closest vectors that it shares the point with). Those are weighted by 1/(edge length). Suppose I have edges, M,N,O,P. Normals A=MxN, B=NxO, C=OxP, D=PxM. So the vertex normal is the normalized vector sum of (A +B)/|N|+(B+C)/|O|+(C+D)/|P|+(A+D)/|M|. That can be rewritten as A(1/|N|+1/|M|) + B(1/|N|+1/|O|) + C(1/|O|+1/|P|) + D(1/|P|+1/|M|) (which is how it was implemented). (It could possibly be even better if it considered the vertex on the other side of the edge for calculating the edge's normal)
Here's the result:



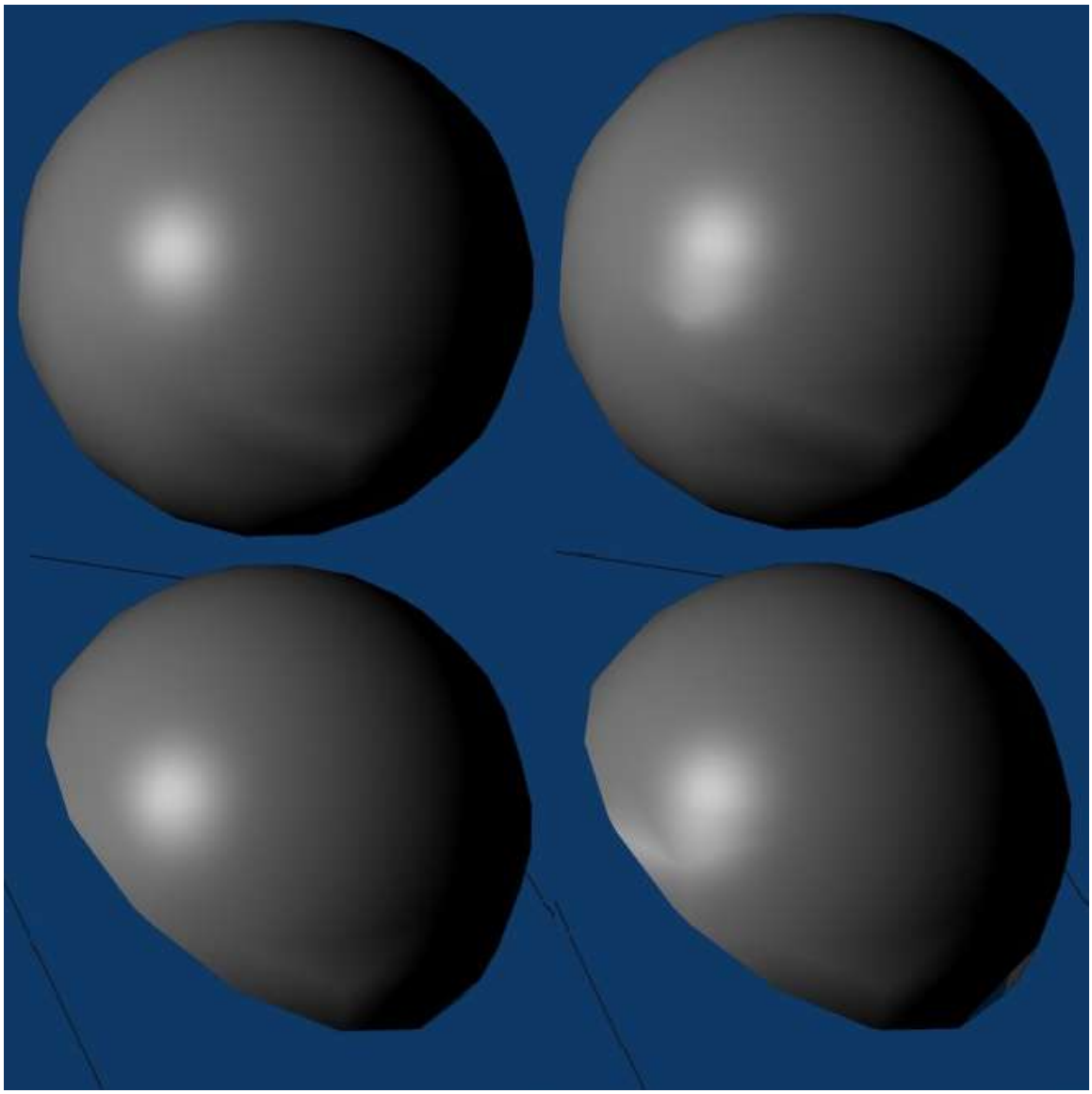
Yay. It works! Mostly, anyway. Much better that the previous two methods. I also tried to multiply it by the $\cos^{-1}$ trick already in place to try to use both ideas. That ended up being worse than either of the two methods. Also, using the square of the lengths, or root-mean-square of the lengths (to avoid calls to sqrt) are short cuts that don't work.

Sum of Vectors:
The previous result is based on weighting vectors by the sum of the inverse of the length of the edges used make them. The idea is shorter edges are more important. But what about the case of wide angles? The edges may be large, but the face is small, and thus needs more weight. But in such a case, the sum of the vectors will be fairly small. And if the angle is not wide, but the edges are short, their sum will also be short. So I did a test using $1/|A + B|$ as the weight for the normal generated be crossing AxB, where A and B are vectors on a face from the vertex in question.



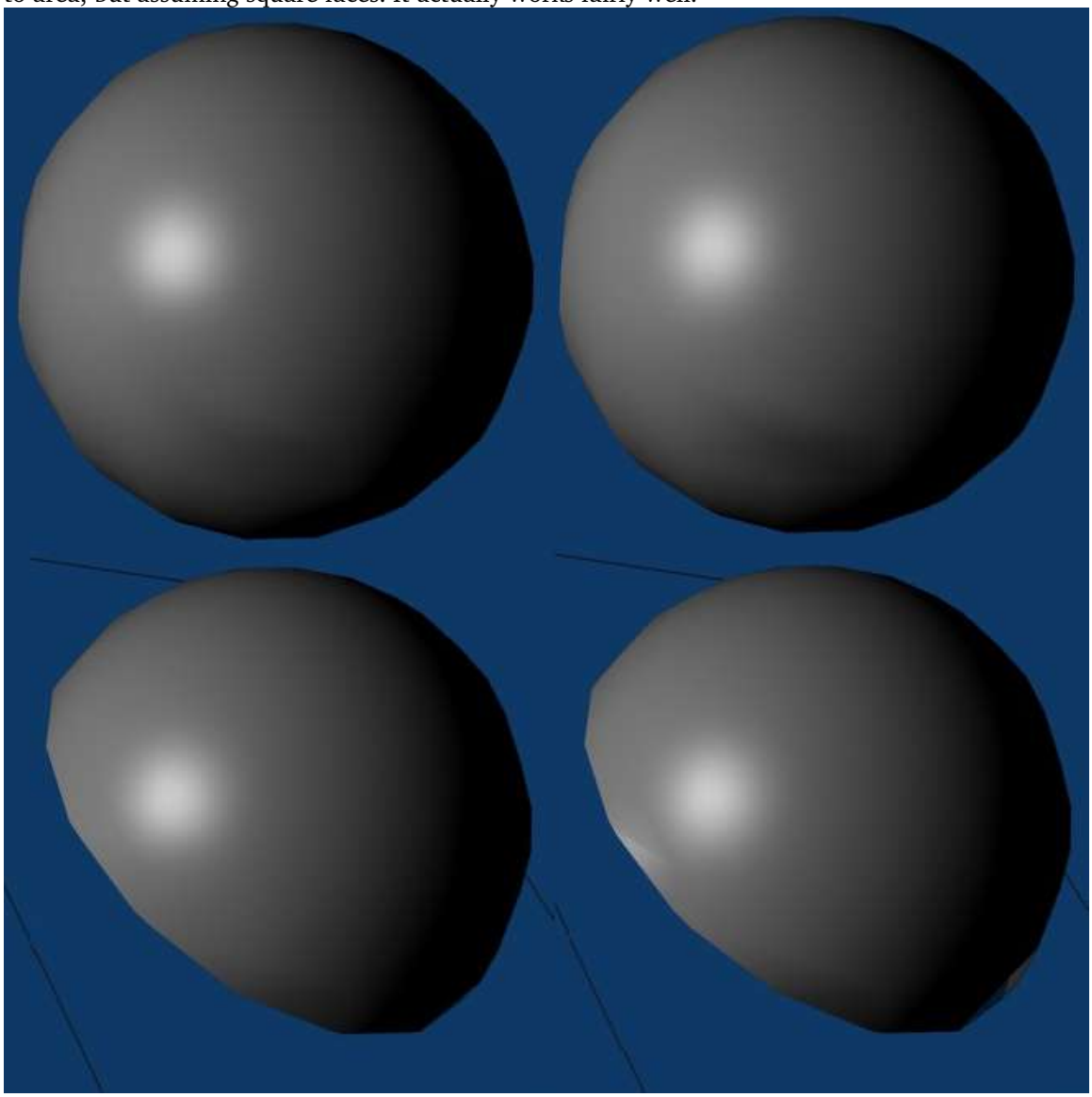
The cut full spheres are now smoother, but the half spheres have more viable artifacts. Apparently the lack of information past the cut line causes problems for this algorithm. Potentially such a situation could be detected and compensated for, but a one-size-fits-all solution would be ideal.

Face Area:
The previous idea is focused on edge lengths. However, while the general idea is sound in 2D, this is 3D. Also, the idea of an edge normal is a little round-about. Really, the edge normal calculation could probably involve some weights as well. So, using area as a weight makes a little more sense. Conveniently, the area of a triangle is half the cross product of two of its edges. Since the cross product must be calculated and normalized, all the needed information is generated regardless. So this costs the least in terms of extra math.



Here every case is fairly smooth, but the shading is a little uneven and awkward.

Multiplication of Inverse Lengths:
This has no real logic. I was just experimenting with random variations. This is a variation on the summing edges method where I multiplied the lengths together. In that sense it is sort of created to area, but assuming square faces. It actually works fairly well.



In the case of the complete spheres, it has the least viable distortion of any of the methods. In the cut spheres there are some problems, but there is the problem of incomplete geometry to guess generate correct normals.

Further Work:

The may vertex normals are calculated in Blender can clearly be improved. However at the moment, I still am undecided what the best method is. It looks like the idea lies somewhere between using edge normals and the area. If anyone has any input, it would be greatly appreciated. (Other techniques, or the most helpful would be a way yo make a good guess for compensating for the chopped off part of the sphere (or other object)) E-Mail me at the address in the beginning of this document. Additionally, I hope to collect models with shading problems to further evaluate the different methods.

I can provide patches if anyone is interested, but my code is a little messy at this point, so I haven't yet.