Tessellation of a Unit Sphere Starting with an Inscribed Convex Triangular Mesh

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/

Created: March 2, 1999
Last Modified: March 2, 2008

Contents

1 Introduction

This document describes how to tessellate a unit radius sphere with a triangular mesh starting with an inscribed convex triangular mesh. A typical starting mesh is an inscribed octahedron with equilateral triangular faces, but the algorithm applies equally well to any inscribed convex triangular mesh. Choice of initial mesh affects the final distribution of triangless. I do not discuss the problem of selecting an initial mesh to produce a final mesh with desired distributional features. *Each triangle in the initial mesh is assumed to have its vertices ordered in a counterclockwise fashion as you look at the triangle from the outside of the polyhedron.*

Each triangle is subdivided into four triangles by computing the midpoints of the triangle edges and connecting them together. The midpoints are not on the unit sphere, but must be moved onto the sphere. Unitizing the points does not necessarily yield a convex polyhedron, especially when the origin is not in the initial convex polyhedron. What I do instead is compute the centroid of the polyhedron (guaranteed to be inside the polyhedron), then project the edge midpoints onto the sphere along the rays connecting the centroid and midpoints.

Subdividing the polyhedron is simple to implement if all you keep track of is a list of triangles and the three vertex locations per triangle. However, if you need more connectivity relationships between the among the vertices, edges, and triangles of the mesh, the implementation becomes more difficult. While simple to describe, the difficult problem is maintaining data structures

2 Data Structures for the Algorithm

The following data types are used. The implementation uses pointers for efficiency by avoiding full copying of the data members.

```
typedef struct
{
    float x, y, z;
}
Point3;

typedef struct
{
    Point3* point;
    int numEdges;  // number of edges sharing the vertex
    struct Edge** edge;  // array of numEdges edge pointers
}
Vertex;

typedef struct
{
    struct Vertex* vertex[2];  // end points of edge
    struct Triangle* triangle[2];  // t[0] and t[1] share the edge
}
Edge;
```

```
typedef struct
{
    struct Vertex* vertex[3];  // vertices of triangle
    struct Edge* edge[3];  // e0 = <v0,v1>, e1 = <v1,v2>, e2 = <v2,v0>
    struct Triangle* adjacent[3];  // adj[i] shares e[i] with triangle
}
Triangle;

typedef struct
{
    int numVertices;
    Vertex* vertex;

    int numEdges;
    Edge* edge;

    int numTriangles;
    Triangle* triangle;

    Point3 centroid;
}
ConvexPolyhedron;
```

While other variations on the data types may be required for an application, this one is used in the algorithm for computing the terminator and silhouette of a convex polyhedron (see the Projection page gr_proj.htm). Goals in the construction are to have no reallocation of memory (avoids fragmentation of the heap) and to update the data structures in-place (efficient use of memory).

Let the initial mesh have ν_0 vertices, ϵ_0 edges, and τ_0 triangles. Let ν_i, ϵ_i, and τ_i be the number of vertices, edges, and triangles, respectively, after the i-th subdivision step. The number of vertices increases by the number of edges. The number of edges is first doubled by the edge splitting, then increased by three times the number of triangles due to connecting the midpoints of the edges. The number of triangles quadruples. The recurrence relations are

$$\nu_{i+1} = \nu_i + \epsilon_i, \;\; \epsilon_{i+1} = 2\epsilon_i + 3\tau_i, \;\; \tau_{i+1} = 4\tau_i, \;\; i \geq 0.$$

While these can be solved in closed form, the code just iterates the equations for the desired number of subdivision steps n to compute the total number of vertices, edges, and triangles required for the final convex polyhedron. The required memory for vertices, edges, and triangles can be allocated all at once for the polyhedron.

The other dynamically allocated quantities are the arrays of edge pointers in the `Vertex` structure. The arrays for the initial vertices never change size. For each added vertex, the number of edges is always six. Because of the edge splitting, the actual edge pointers may change during a subdivision step.

3 Subdivision Algorithm

3.1 Centroid of Polyhedron

Initially, and after each subdivision step, the centroid of the convex polyhedron is computed to be used in the midpoint projection phase of the subdivision. This point is computed as the average of the current vertices in the polyhedron. The centroid at step s is

$$\mathbf{C} = \sum_{j=0}^{\nu_s} \mathbf{P}_j$$

where the vertex locations are $\mathbf{P}_0$ through $\mathbf{P}_{\nu_s-1}$.

3.2 Intersection of Ray with Sphere

A new vertex is added per edge by computing the midpoint $\mathbf{M}$ of the edge, then projecting that point along the ray starting at the centroid $\mathbf{C}$ and passing through $\mathbf{M}$. If edge E has endpoints $\mathbf{P}_0$ and $\mathbf{P}_1$, then $\mathbf{M} = (\mathbf{P}_0 + \mathbf{P}_1)/2$. The ray is given parametrically as $\mathbf{X}(t) = \mathbf{C} + t(\mathbf{M} - \mathbf{C})$ where $t \geq 0$. Since both $\mathbf{C}$ and $\mathbf{M}$ are inside the sphere, $\mathbf{X}(t)$ must be inside the sphere for any $t \in [0, 1]$. The new vertex location occurs where the ray and sphere intersect. If $\bar{t}$ is the parameter value at the intersection, then $\bar{t} > 1$ and the squared length of $\mathbf{X}(\bar{t})$ is 1. This condition is a quadratic equation

$$\begin{aligned} 1 &= \mathbf{X}(\bar{t}) \cdot \mathbf{X}(\bar{t}) \\ &= [\mathbf{C} + \bar{t}(\mathbf{M} - \mathbf{C})] \cdot [\mathbf{C} + \bar{t}(\mathbf{M} - \mathbf{C})] \\ &= \mathbf{C} \cdot \mathbf{C} + 2\mathbf{C} \cdot (\mathbf{M} - \mathbf{C})t + (\mathbf{M} - \mathbf{C}) \cdot (\mathbf{M} - \mathbf{C})t^2. \end{aligned}$$

If $\mathbf{D} = \mathbf{M} - \mathbf{C}$, the quadratic equation is $a_2t^2 + a_1t + a_0 = 0$ where $a_2 = \mathbf{D} \cdot \mathbf{D}$, $a_1 = 2\mathbf{C} \cdot \mathbf{D}$, and $a_0 = \mathbf{C} \cdot \mathbf{C} - 1$. As argued earlier, the equation has one root larger than 1, namely

$$\bar{t} = \frac{-a_1 + \sqrt{a_1^2 - 4a_0a_2}}{2a_2}.$$

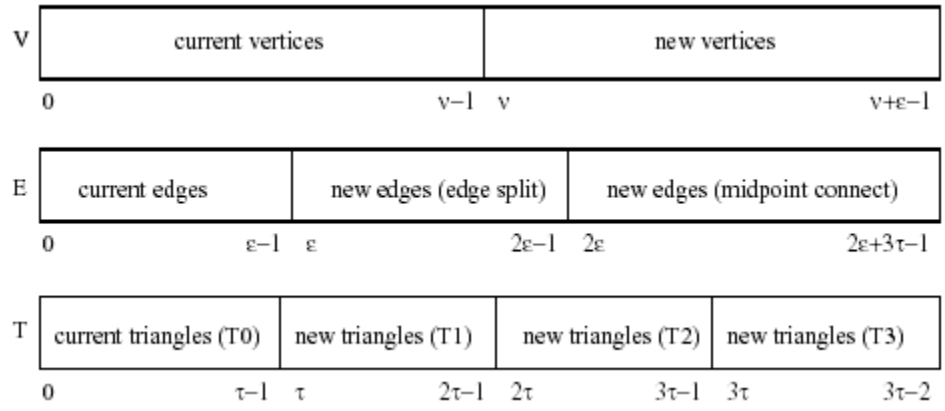
The location for the new vertex is therefore $\mathbf{P} = \mathbf{C} + \bar{t}\mathbf{D}$.

3.3 Memory Organization

The total amount of memory required for the vertex, edge, and triangle arrays in the final polyhedron is computed and the corresponding memory is allocated. The vertex array has ν_n elements, the edge array has ϵ_n elements, and the triangle array has τ_n elements. The initial items are stored at the beginning of each of the arrays.

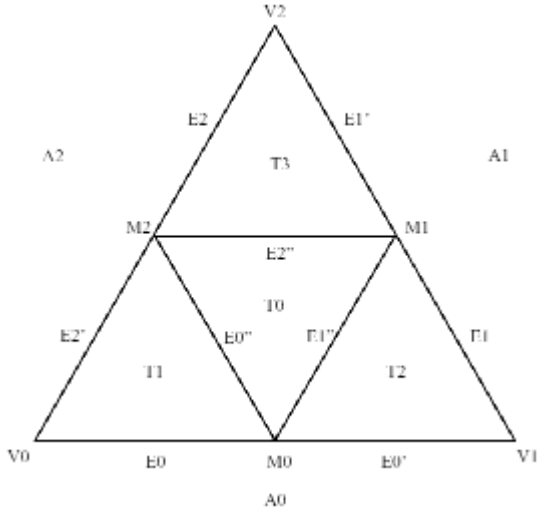
For a single subdivision step, the working set in memory is shown in Figure 3.1.

Figure 3.1 Working set of vertices, edges, and triangles.



The values ν, ϵ, and τ are the current numbers of vertices, edges, and triangles, respectively. The partitioning of the triangle subarray into four sections corresponds to splitting the triangle into four pieces as shown in Figure 3.2.

Figure 3.2 Subdivided triangle.



If the figure represents the triangle stored at the t-th location in the array, then the subtriangle T_k is stored at location $t + k\tau$ for $k = 1, 2, 3$. The subtriangle T_0 is stored at the same location as the original triangle. This in-place storage requires careful bookkeeping to avoid overwriting old triangle information with new information before the old information is no longer needed for other parts of the algorithm.

3.4 Pseudocode for a Single Subdivision

As the data structures are updated, it is important to retain old vertex, edge, or triangle information that is necessary for use at a later stage of the update. In the pseudocode, variables `vmax`, `emax`, and `tmax` correspond to ν, ϵ, and τ, the current number of vertices, edges, and triangles. The variables `vertex`, `edge`, and `triangle` refer to the arrays V, E, and T shown in Figure 1. The variable `centroid` is the centroid of the current polyhedron.

The first two safe operations are to add the new vertex locations and to split the edges.

```
for (e = 0; e < emax; e++)
{
    // generate new vertex M from edge[e]
    E0 located at edge[e];
    P0 = E0.vertex[0].point;
    P1 = E0.vertex[0].point;
    mid = (P0+P1)/2;
    M located at vertex[vmax+e];
    M.point = RaySphereIntersection(centroid,mid);
    M.numberOfEdges = 6;
    M.edge = allocate 6 edge pointers;

    // split edge E0 = <V0,V1> into E0 = <V0,M> and E1 = <V1,M>
    E1 located at edge[emax+e];
    E1.vertex[0] = E0.vertex[1];
    E0.vertex[1] = M;
    E1.vertex[1] = M;
}
```

The edge pointers for the new vertices are set later to point to the subdivided edges. The triangle pointers for the split edges are the old ones and are updated later to point to the subdivided triangles. Once the new vertex locations are known, the centroid of the old and new vertices can be precomputed for the next subdivision pass.

An iteration can now be made over the current triangles to update various fields in the data structures. The function `VertexIndex(V)` returns the index of vertex V in the array of vertices. Similarly `EdgeIndex(E)` returns the index of edge E in the array of edges and `TriangleIndex(T)` returns the index of triangle T in the array of triangles.

The code uses four indices into the triangle array, `t0` for the current triangles (then later to represent the middle triangle in the subdivision), and `t1`, `t2`, and `t3` for the other subdivided triangles. Initially

```
t0 = 0;
t1 = tmax;
t2 = 2*tmax;
t3 = 3*tmax;
```

In the following discussion, each displayed block of code is contained in a loop where `t0` varies from `0` to

tmax-1. All four counters are incremented each pass of the loop. The first displayed block of code indicates the relationships that are immediately discernable from Figure 2.

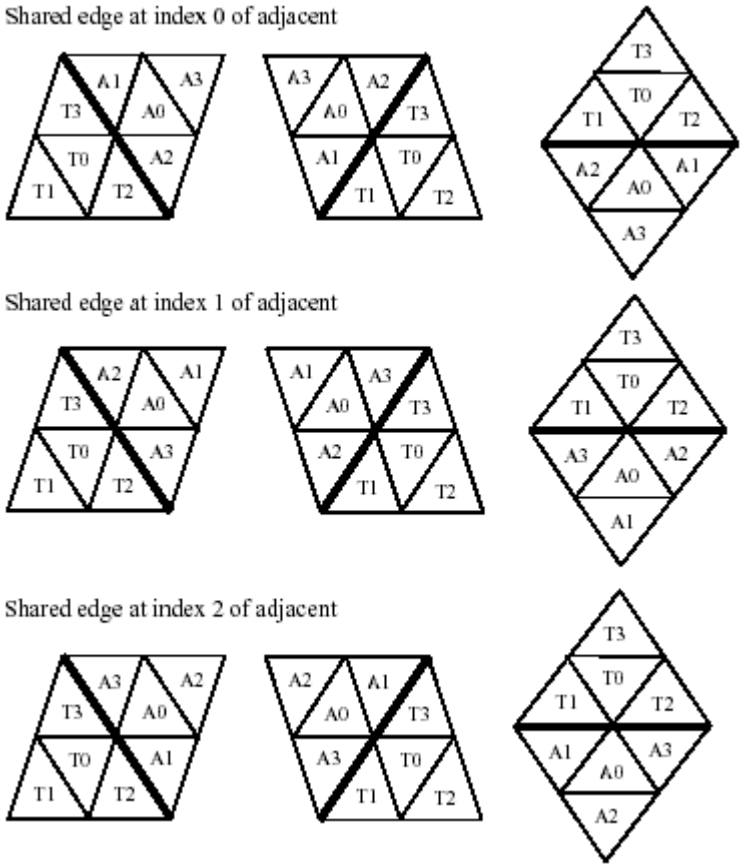
```
// get triangles to process
Tj located at triangle[tj] for j = 0,1,2,3;
Vi = T0.vertex[i] for i = 0,1,2;  // vertices
Ei = T0.edge[i] for i = 0,1,2;  // original edges
Ei' is other half of original edge formed by split;
Ai = T0.adjacent[i] for i = 0,1,2;  // adjacent triangles
Mi located at vertex[vmax+EdgeIndex(Ei)] for i = 0,1,2;  // midpoints

// edges corresponding to connections of midpoints
Ei" located at edge[2*emax+3*t0+i] for i = 0,1,2;

// set vertices and triangles for edges
E0".vertex[0] = M2;
E0".vertex[1] = M0;
E0".triangle[0] = T0;
E0".triangle[1] = T1;
E1".vertex[0] = M0;
E1".vertex[1] = M1;
E1".triangle[0] = T0;
E1".triangle[1] = T2;
E2".vertex[0] = M1;
E2".vertex[1] = M2;
E2".triangle[0] = T0;
E2".triangle[1] = T3;

// set some members of triangle T1
T1.vertex[0] = V0;
T1.vertex[1] = M0;
T1.vertex[2] = M2;
T1.edge[0] = E0;
T1.edge[1] = E0";
T1.edge[2] = E2';
T1.adjacent[1] = T0;

// set some members of triangle T2
T2.vertex[0] = M0;
T2.vertex[1] = V1;
T2.vertex[2] = M1;
T2.edge[0] = E0';
T2.edge[1] = E1;
T2.edge[2] = E1";
T2.adjacent[2] = T0;

// set some members of triangle T2
T3.vertex[0] = M2;
```

```
T3.vertex[1] = M1;
T3.vertex[2] = V2;
T3.edge[0] = E2";
T3.edge[1] = E1';
T3.edge[2] = E2;
T3.adjacent[0] = T0;
```

The subdivided triangles `T1`, `T2`, and `T3` have some adjacent triangles that are themselves subdivided triangles of adjacent triangles of the original triangle `T0`. At this point in the algorithm we know the index of each original adjacent triangle in the triangle array for the polyhedron. We can use this index to locate the triangle array elements for the subdivided triangle of the adjacent triangle. However, there is one complication. Figure 2 shows a triangle in a *standard* configuration based on how the vertices and edges were labeled (vertices start at lower left corner of triangle, others labeled in counterclockwise order). An adjacent triangle can have one of three possible orientations in its relationship to the central triangle. The code must determined which edge of the adjacent triangle is shared with the central triangle. This information allows proper labeling of the adjacency relationship between subdivided triangles between adjacent original triangles.

Figure 3.3 shows the nine cases to consider, three choices for adjacent triangle and three orientations for that triangle.

Figure 3.3 Possible orientations of adjacent triangle with central triangle.



The pseudocode for handling the adjacent triangle `A0` is

```
// get the indices for the subdivided triangles of A0
i0 = TriangleIndex(A0);
i1 = i0+tmax;
i2 = i1+tmax;
i3 = i2+tmax;

orient = shared ege of A0 and T0 is A0.edge[orient];
if ( orient == 0 )
{
    T1.adjacent[0] is triangle[i2];
    T2.adjacent[0] is triangle[i1];
    triangle[i2].adjacent[0] is T1;
    triangle[i1].adjacent[0] is T2;
```

```
}
else if ( orient == 1 )
{
    T1.adjacent[0] is triangle[i3];
    T2.adjacent[0] is triangle[i2];
    triangle[i3].adjacent[1] is T1;
    triangle[i2].adjacent[1] is T2;
}
else
{
    T1.adjacent[0] is triangle[i1];
    T2.adjacent[0] is triangle[i3];
    triangle[i1].adjacent[2] is T1;
    triangle[i3].adjacent[2] is T2;
}
```

Similar blocks of code are in the source file for adjacent triangles `A1` and `A2`.

Finally in this loop, the edge pointers for the new vertices can be set. The real code initializes the number of edges of each new vertex to zero as a flag. The first time a new vertex is encountered, it is processed as a midpoint of an edge of a triangle. At this time four of the edges sharing the vertex are known. At at later pass through the loop, the adjacent triangle sharing the edge whose midpoint generated the new vertex. At this time the remaining two edge pointers can be set. For example,

```
// add edge links to midpoint vertices
if ( M0.numEdges == 0 )
{
    // add four edges (first time edges are added for this vertex)
    M0.numEdges = 4;
    M0.edge[0] is T1.edge[0];
    M0.edge[1] is T1.edge[1];
    M0.edge[2] is T2.edge[2];
    M0.edge[3] is T2.edge[0];
}
else if ( M0.numEdges == 4 )
{
   // add two edges (last time edges are added for this vertex)
   M0.numEdges = 6;
   M0.edge[4] is T1.edge[1];
   M0.edge[5] is T2.edge[2];
}
```

Similar blocks of code are in the source file for new vertices `M1` and `M2`.

A second iteration over the current triangles is made. This step requires information produce by the first iteration over all triangles. The following blocks of pseudocode are contained in a loop over the four triangle counters, just as before. The initial values of the counters are

```
t0 = 0;
```

```
t1 = tmax;
t2 = 2*tmax;
t3 = 3*tmax;
```

The first part of the loop is

```
// get triangles to process
Tj located at triangle[tj] for j = 0,1,2,3;
Ei = T0.edge[i] for i = 0,1,2;
Mi = vertex[vmax+EdgeIndex(Ei)] for i = 0,1,2;

// set vertices, edges, adjacencies for middle triangle
T0.vertex[0] is M2;
T0.vertex[1] is M0;
T0.vertex[2] is M1;
T0.edge[0] is T1.edge[1];
T0.edge[1] is T2.edge[2];
T0.edge[2] is T3.edge[0];
T0.adjacent[0] is T1;
T0.adjacent[1] is T2;
T0.adjacent[2] is T3;
```

The subdivided triangle edge pointers can be set at this time. For example, at `T1`,

```
T1.edge[0].triangle[0] is T1;
T1.edge[0].triangle[1] is T1.adjacent[0];
T1.edge[2].triangle[0] is T1;
T1.edge[2].triangle[1] is T1.adjacent[2];
```

The actual code initializes the triangle pointers to zero and tests before the pointers are set. Otherwise, each triangle pointer is written twice (simply an efficiency gain). Similar blocks of code exist for `T2` and `T3`.

The last step is to adjust the edge pointers of the original vertices to account for the edge splitting. The main concern is to select the correct half edge to store a pointer to. This can be determined by comparing the vertex pointer values of the half edges to the testing vertex.

```
for (v = 0; v < vmax; v++)
{
    V is vertex[v];
    for (e = 0; e < V.numEdges; e++)
    {
        E is V.edge[e];
        if ( E.vertex[0] not equal V and E.vertex[1] not equal V )
        {
            // V.edge[e] currently points to wrong half edge, switch it
            V.edge[e] = edge[emax+EdgeIndex(E)];
        }
    }
```

4 The Test Program

The test program is a Microsoft Windows application that allows you to subdivide either a tetrahedron or an octahedron. The tessellated objects may be rotated via keystrokes. The number of subdivisions is also controlled by keystrokes. The tessellated objects are displayed as wireframes in the center of the window. It is possible to view either a parallel or a perspective projection of the objects.