Triangulation of a Sphere

Ryan S. Shendler

Department of Computer Science, Binghamton University

CS460: Computer Science

Dr. Lijun Yin

May 14, 2022

Abstract

Triangulation is the process of representing arbitrary 3D models as a mesh of triangular faces. Triangulation is important because most graphics systems are optimized for drawing triangles. Currently, one of the most popular triangulation methods is Marching Triangles. Marching Triangles utilizes Delaunay Triangulation in order to generate an evenly distributed mesh of equilateral triangles. However, Marching Triangles is mathematically complex and computationally expensive. This report proposes a new triangulation algorithm that is simpler and less computationally demanding than Marching Triangles. Preliminary research shows that this new algorithm requires significantly less computation than Marching Triangles.

Introduction

Most modern graphics systems optimize the process of rendering triangles. This is because triangles are the polygons with the fewest number of vertices and are therefore the simplest polygons to render. Additionally, it is possible to represent any other polygon as a mesh of connected triangles. This can be extended even further to representing entire 3D models as triangular meshes. The process of developing a triangular mesh that represents a given 3D model is known as triangulation. In this report, we will be discussing the process of triangulating a sphere.

Literature Review

Currently, one of the most popular triangulation algorithms is the Marching Triangles method, which is based on the Delaunay Triangulation. The Delaunay Triangulation takes a set of points P, which will make up the vertices of the triangles, and returns a triangulation DT(P) such that no point in P is within the circumcircle of any triangle in DT(P) (Delaunay triangulation, 2022). This triangulation method is popular because it maximizes the minimum angle of each triangle, which results in a triangular mesh where the triangles are evenly distributed and are of similar size and shape. This is in contrast with some other triangulation methods, which may create very thin "sliver triangles" in the gaps between normally-sized triangles.

In order to triangulate a sphere using Marching Triangles, the user needs to provide a seed point s, which is located near the sphere surface (Hartmann, 1998, p. 97). The algorithm then finds surface point p_1, which is the surface point closest to s. This surface point is found by using Newton's method to incrementally search for the closest surface point to s (Hartmann, 1998, p. 98). Next, the algorithm creates a hexagon in the tangent plane centered on p_1, with

points $q_2 \ldots q_7$ being the vertices on this hexagon. Finally, the algorithm finds surface points $p_2 \ldots p_7$ which correspond to $q_2 \ldots q_7$. The surface hexagon $p_2 \ldots p_7$ is called the actual front polygon, and is denoted as Π_0.

After creating Π_0, the algorithm next tries to expand Π_0. In order to expand Π_0, the algorithm first calculates the front angle for each point in Π_0. The front angle of a point is the angle of the area to be triangulated if Π_0 was to be expanded from that point (Hartmann, 1998, p. 97). Once the front angles have been calculated, the algorithm then checks to see if each point p_i in Π_0 is near another point in Π_0 other than p_i and its neighbors or any point in another front polygon Π_k. If p_i is near another point in Π_0, then the algorithm splits Π_0 into two separate front polygons. Otherwise, if p_i is near a point in front polygon Π_k, then the algorithm merges Π_k into Π_0, deleting Π_k in the process (Hartmann, 1998, p. 97). Finally, the algorithm selects the point p_m with the smallest front angle and expands Π_0 from that point. This expansion is done by surrounding p_m with approximately equilateral triangles, then deleting p_m from Π_0 and adding the new points to Π_0 (Hartmann, 1998, p. 97). This process of calculating front angles, merging or splitting front polygons, and then expanding Π_0 is repeated until Π_0 only has three points left. Once this occurs, render a triangle using these three points as the vertices and then select a new front polygon to expand. The triangulation is complete when there are no more front polygons remaining.

The main advantage of Marching Triangles over other triangulation methods is it creates an evenly distributed mesh of equilateral triangles with little to no sliver triangles. Another advantage of Marching Triangles is that it is a generalized algorithm that can be used to triangulate any shape given that shape's implicit surface formula. This means that Marching Triangles can create triangular meshes for spheres, cylinders, toruses, and other 3D shapes.

However, Marching Triangles also has several downsides, the largest being the significant amount of computational overhead required. For a single iteration of front polygon expansion, Marching Triangles has to compute the front angle for every point in Π_0, the distance between all pairs of points in Π_0, the distance between points in Π_0 to points in any other front polygon, and the vertices for each of expanded triangles, as well as their corresponding surface points. Another issue with Marching Triangles is that it requires a seed point s, which means the user needs to have some way of determining a suitable seed point in order for the algorithm to work. Finally, Marching Triangles is a mathematically-complex algorithm and can be difficult for new students to understand and implement. For these reasons, this report proposes a new sphere triangulation algorithm that is simpler and less computationally expensive than Marching Triangles.

Algorithm

My proposed algorithm triangulates a sphere of a given radius that is centered at the origin. The only computation required for my algorithm is calculating the coordinates of the vertices. My algorithm generates these vertices using spherical coordinates. The spherical coordinate system is an alternative to the traditional three-dimensional Cartesian coordinate system which can be used to describe points on the surface of a sphere (Spherical coordinates, 2022). A point in the spherical coordinate system is described by three parameters: r, Θ, and φ. The parameter r is the radius of the sphere. The parameter Θ is the angle between the point and the y-axis while φ is the angle between the point and the z-axis. The angle Θ is constrained to the range $[0, \pi]$, while φ is constrained over the range $[0, 2\pi)$. It is possible to convert a point from spherical coordinates to Cartesian coordinates by using the following formulas: $x = r\sin\Theta\sin\varphi$, $y = r\cos\Theta$, and $z = r\cos\varphi\cos\Theta$.

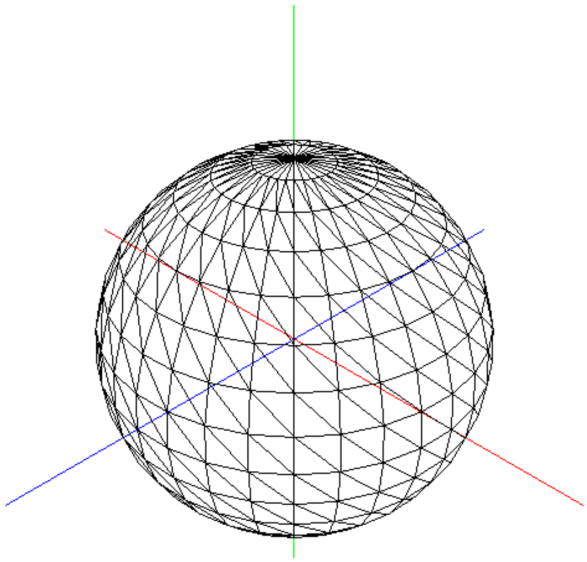
My algorithm has three main steps: generating vertices, triangulating the polar region of the sphere, and triangulating the non-polar region of the sphere. In order to generate the vertices of the mesh, my algorithm utilizes the spherical coordinate formulas to generate vertices along the lines of latitude of the sphere. This is performed by iterating over Θ from 0 to φ and iterating over φ from 0 to 2π. For each step in this iteration, I use the formulas x = rsinΘsinφ, y = rcosΘ, and z = rcosφcosΘ to calculate the (x, y, z) coordinates for the vertex corresponding to the current values of Θ and φ. The radius r of the sphere remains fixed throughout this entire process. The step value for this iteration is $\frac{\pi}{interval}$ where interval is a global variable with a default value of 16. Users can increase the size of interval in order to generate more vertices or decrease its size to generate fewer vertices.

Once the algorithm has generated all of the vertices, it proceeds to render the triangular mesh for the polar regions of the sphere. This is performed by using the GL_TRIANGLE_FAN drawing option in OpenGL, which draws triangles radiating out from a single central point. My algorithm uses the polar points as the central point and then draws the remaining vertices from the northernmost or southernmost line of latitude. This results in a circular mesh of triangles, which all share a vertex at the pole of the sphere.

In order to render the non-polar region of the sphere, my algorithm creates a rectangular mesh using the generated vertices. My algorithm then splits each rectangle in this mesh along its diagonal, thereby transforming each rectangle into a pair of right triangles. Figure 2 depicts a sphere triangulation created by my algorithm.

Figure 1

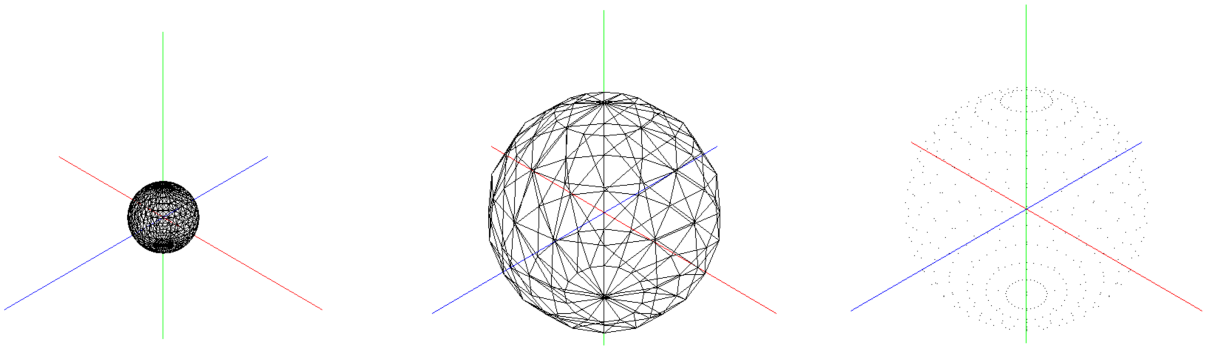
Sphere Triangulation Created By My Algorithm



As previously mentioned, my algorithm allows a user to change the value of the interval variable in order to generate more or fewer vertices. This allows users to increase or decrease the detail of the sphere, respectively. Additionally, users can also vary the radius of the sphere. These options are available to the user through a menu click. Users can also choose one of three display modes for the triangulated sphere: points, wireframe, or backface culling. The points mode only displays the vertices, while the wireframe mode displays the actual triangular mesh. Backface culling is similar to the wireframe mode but with backface culling enabled. This mode makes it easier to see the structure of the mesh. Figure 2 depicts these different display options.

Figure 2

From Left to Right: Smaller Radius, Larger Interval/Fewer Vertices, Points Display Mode



Results

Table 1 displays the results of using my algorithm to triangulate a unit sphere with different step values. Compare this with Table 2, which displays the results of the using Marching Triangles to triangulate a unit sphere using different step values (Kronemijer, 1998, p. 15). The step value for Marching Triangles is the approximate edge size of each triangle. A smaller step size results in smaller triangles, which in turn results in a more detailed triangulation.

Table 1

Triangulating a Unit Sphere with my Algorithm

Step Value	**Number of Vertices**	**Number of Faces**
$\pi/4$	26	48
$\pi/8$	114	224
$\pi/16$	482	960
$\pi/32$	1986	3968
$\pi/64$	8066	16128

Table 2

Triangulating a Unit Sphere using Marching Triangles

Surface	sphere	Sphere	sphere
Delta	0.20	0.10	0.05
Triangles	768	3034	11830
Edges	1152	4551	17745
Vertices	385	1518	5916
maximum number of front edges	41	81	158
f evaluations	1540	4554	17748
f evaluations per vertex	4.0	3.0	3.0
∇f evaluations	1926	6073	23665
∇f evaluations per vertex	5.0	4.0	4.0
$max \lvert f(vertex) \rvert$	2.37e-16	2.57e-16	2.81e-16
minimal triangle angle	29.1	23.8	31.7
maximal triangle angle	101.0	122.0	51.5
average minimal angle	51.2	51.4	93.9
average maximal angle	70.6	70.6	70.1
total area	12.461	12.540	12.559
average triangle area	0.0162	0.0041	0.0011
minimal triangle area	0.0084	0.0025	0.0006
maximal triangle area	0.0338	0.0097	0.0020

Note that $\pi/16$ is approximately 0.2, $\pi/32$ is approximately 0.1 and $\pi/64$ is approximately 0.05. Although the step values have different meanings in my algorithm and Marching Triangles, they still have a similar purpose. If we equate the step values in my algorithm to those in Marching Triangles, we can see that my algorithm consistently required fewer vertices and faces to triangulate the unit sphere. Additionally, recall that the only computation required in my requirement is the calculation of the vertex coordinates. Therefore, for my algorithm, the number of computations is equal to the number of vertices. In contrast, the number of computations in Marching Triangles is the sum of the f evaluations and ∇ f evaluations. This means that Marching Triangles required significantly more computations to triangulate the unit sphere for all step values.

Conclusion

From my preliminary research, I can conclude that my algorithm is able to triangulate a unit sphere with fewer vertices and faces, and with less computational overhead, than the Marching Triangles algorithm. However, this comparison is quite limited at the moment, as my data for the performance of Marching Triangles is taken from a study by E.R. Kronemijer, and

not from my own implementation of Marching Triangles (1998, p.15). In the future, I would like to develop my own implementation of Marching Triangles and compare it with my proposed algorithm. Some factors I would like to compare are the time taken by each algorithm to triangulate a sphere, the number of vertices and faces in the triangulation, and the number of computations required to create each triangulation.

Another issue with my triangulation is that the triangular mesh is not evenly distributed, which can cause sliver triangles to form around the poles. As previously mentioned, the Delaunay Triangulation used by Marching Triangles does not suffer from this issue, which is a huge benefit for Marching Triangles over my algorithm. In the future, I would like to research methods of making my mesh more evenly distributed.

References

Hartmann, E. (1998). A marching method for the triangulation of surfaces. *The Visual Computer, 14*, 95-108.

Kronemijer, E. R. (1998). *Triangulation of implicit surfaces* [Master's Thesis, University of Groningen].

Spherical coordinates. from Wolfram MathWorld. (n.d.). Retrieved May 14, 2022, from https://mathworld.wolfram.com/SphericalCoordinates.html

Wikimedia Foundation. (2022, May 12). *Delaunay triangulation*. Wikipedia. Retrieved May 14, 2022, from https://en.wikipedia.org/wiki/Delaunay_triangulation