Computer-Aided Design and Applications

http://www.cadanda.com

Computations on an Ellipsoid for GIS

James R. Miller[1] and Tom Gaskins[2]

[1]University of Kansas, jrmiller@ku.edu
[2]NASA, tom@tomgaskins.com

ABSTRACT

We describe several challenging issues related to virtual globe manipulations for GIS that highlight the important interplay between geometric modeling and scientific visualization. We begin by motivating and describing a globe tessellation algorithm designed to avoid singularities at the poles, then discuss other uses of the tessellation that enable certain types of shape modeling on the surface of the virtual globe.

Keywords: level of detail modeling; Web-based retrieval, analysis and visualization.
DOI: 10.3722/cadaps.2009.575-583

1. INTRODUCTION

Modern GIS systems employ a mix of geometric modeling, visualization, and distributed data management. Elevation models, subsurface structures, and other geographic objects are oftentimes modeled using a variety of 3D representations. NASA World Wind [9] is an open source Java SDK that allows users to create standalone applications in which their data and models can be presented in the context of a multi-resolution model of a globe (e.g., the earth, the moon, mars). Controls allow users to rotate, zoom, pan and navigate throughout the environment. The SDK handles globe geometry generation with appropriate elevation models as well as texture mapping which can be used to present realistic surface detail. Fig. 1 illustrates the earth at two different scales and viewing angles.

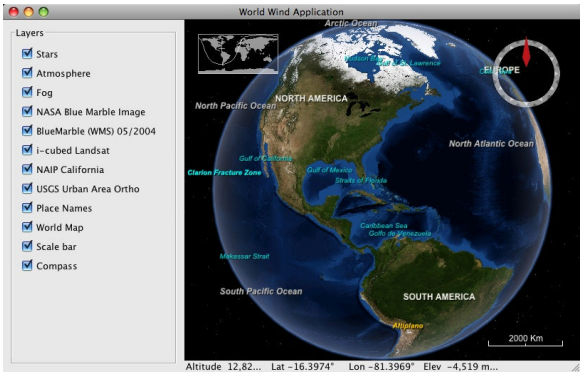

Fig. 1(a): View of the Earth from space.

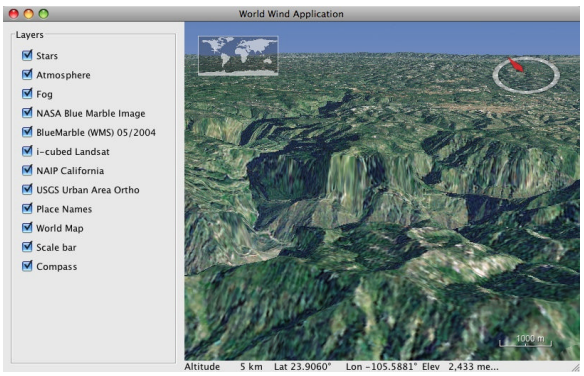

Fig. 1(b): Zoomed in with a low line of sight.

In this paper, we describe some internal modeling and analysis algorithms that highlight the multi-resolution character of the system and demonstrate the interplay between geometry modeling,

visualization, and distributed data management in a GIS environment. Some of the algorithms and structures are new; others are established algorithms applied in a novel context to support the need for high frame rate navigation through a large and complex 3D environment.

NASA World Wind is similar in appearance to several other systems. Google Earth [5] is one of the better known examples. Unlike World Wind, Google Earth is a standalone application, and it is not open source. The basic binary version can be downloaded for free from the web site; an advanced version is available for purchase. User data can be incorporated into the system in a variety of ways, most notably using KML [8].

Microsoft's Virtual Earth [7] runs in a web browser and is a commercial product. User data can be imported using formats like KML. It is also possible to add functionality to the system using tools such as Java or JavaScript, provided they conform to relevant standards such as SOAP.

Finally, ESRI's ArcGIS Explorer [2] is also a standalone application available via a free binary download and is not open source. While all the other systems are cross-platform applications, ArcGIS Explorer runs only on MS Windows platforms, requiring .NET and Internet Explorer. User data can be imported from any of the other ArcGIS products.

An important characteristic distinguishing World Wind from all the others mentioned is the fact that it is not only open source, but also designed so as to enable users to develop their own standalone applications, augmenting or even completely replacing the demonstration user interface seen in the example applications. While a variety of sample applications are included in the download as a guide, it is straightforward for users to develop their own look and feel, and to add their own specialized features. Only a few simple conventions are required to tap into the internal engine. Graphics are generated using Java OpenGL (JOGL) [4], and interactive controls are developed using Java Swing.

Central to the geometry modeling portion of the engine is the ability to tessellate the globe based on the current line of sight and field of view. Typically only that portion of the globe within the field of view is fully tessellated. A proper tessellation, including relevant elevation information, is critical for several reasons. Image data is imported and applied to the tessellated geometry using texture maps. Other shapes and computations are defined as being applied to a region of the surface and must be appropriately rendered or computed regardless of the zoom level. To avoid doing too much or too little computation, it is convenient to render shapes (and/or do analysis) using an extracted subset of the tessellation. Much of the rest of this paper expands upon these criteria and presents algorithms designed to meet the needs.

2. GLOBE TESSELLATION ISSUES

The globe rendering process begins by assuming an idealized ellipsoid representation of a planetary body such as the earth, moon, or mars. Since interactive graphics APIs like OpenGL require planar polygons, the ellipsoid must be tessellated. This seemingly simple problem has actually drawn a considerable amount of research attention. The tessellation must be locally refined as the user rotates, zooms, and pans, hence a recursive tessellation algorithm which produces polygons whose size is a function of recursive depth is common. Given that we also want to augment the idealized ellipsoid using an elevation model (actually several) to represent surface geography at an appropriate resolution as we zoom and pan, the tessellation serves a dual purpose by providing a vehicle to linearly interpolate elevation values between the grid points at which they are defined.

While we ultimately want triangles for the actual rendering, an initial tessellation based on quadrilaterals is typically created. As we mention below, this not only leads to a simpler and more efficient tessellation algorithm, but it is also easier to control certain important global characteristics of the tessellation. We can now summarize several important tessellation requirements.

We need to map very general types of auxiliary data onto our tessellated globe. Georeferenced images are texture mapped onto the tessellated triangles. Other geometric shapes, typically defined in GIS

latitude-longitude coordinates, are also common. Hence we want our tessellation to be easily related to these auxiliary georeferenced data.

It is important that the tessellation algorithm produce triangles at a given level of subdivision that have roughly similar aspect ratios, sizes, and shapes to optimize the quality of the image, better manage the overall recursive subdivision process, and avoid numerical problems related to thin sliver triangles. Moreover, in order to avoid significant special case handling in several internal algorithms, we want to prevent creating triangles that cross the International Date Line.

Finally, it is convenient – though not strictly necessary – if triangle edges are aligned with lines of constant latitude or longitude. Clearly this cannot be the case for all triangle edges. The algorithms we describe below will generally produce two such edges per triangle, except for most areas in the polar regions.

The obvious solution is an initial tessellation based on generating quadrilaterals whose edges are lines of constant latitude and longitude. While this works well over much of the globe, it leads to many problems in the region of the poles. There are numerical issues caused by the fact that quads get very tall and skinny, and quads that actually touch the pole have an edge that degenerates to a point. An even greater problem is that many more tiles are in view at the poles than there are at lower latitudes for a given level of subdivision, thus leading to the generation of a huge number of very small triangles. This is a problem in the context of texture mapping (mapping raster images to geometric shapes like triangles and other polygons). Like its contemporaries, World Wind uses texture mapping to display resolution-dependent imagery such as oceans, mountains, rivers, and other details, depending on the current zoom level. OpenGL allows multiple versions of a desired texture to be defined at various resolutions so that the system can automatically retrieve the version of the texture whose resolution is the best match possible for the current screen space size of the polygon being rendered. This technique is known as mip-mapping. Having very tall skinny triangles is well-known to have a very adverse impact on texture mapping in general, and mip-mapping in particular. Let us simply bear the polar issues in mind for now as we review some alternative tessellation approaches.

One conceptually simple algorithm employs three orthogonal families of planes – one perpendicular to each of the three Cartesian axes – slicing the ellipsoid. While it is conceptually simple and produces no degeneracies or other anomalous behavior at the poles, it exhibits several fatal flaws. First, there is not a very natural way to define how to localize areas for refined recursive subdivision. More significantly, it produces polygons that have highly irregular topologies, shapes, and sizes at each level of subdivision. Some regions are even concave. (See Fig. 2.) It is these undesirable characteristics that also make this seemingly simple solution very difficult to implement reliably.

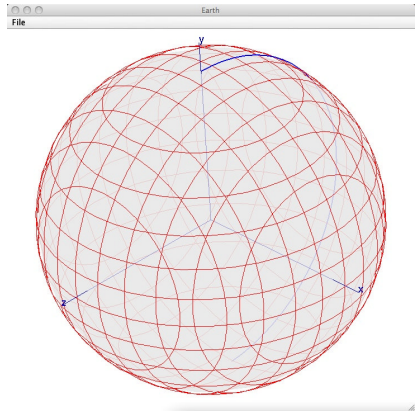

Fig. 2: Three orthogonal families of plane slices. The blue line leaving the North Pole is the International Date Line.

One common ellipsoidal tessellation algorithm begins by inscribing an icosahedron inside the ellipsoid as shown in Fig. 3(a). The various faces can be individually subdivided recursively, producing triangles that are very nearly equilateral everywhere. (See Fig. 3(b).) Teanby [6] has successfully used this tessellation as the basis for binning of remote sensing data. Unfortunately, this tessellation unavoidably generates triangles that cross the International Date Line as can be seen in Fig. 3(b). While triangles crossing the date line could be split, forming a pair of right triangles, it was our experience when doing this in World Wind that special case treatment was required for the right triangles in certain algorithms. However an even more significant issue relates to all triangles generated by the icosahedral subdivision algorithm. Our overall computational efficiency depends significantly on bounding sector schemes. A sector is defined as a rectangle in lat-lon space. What we have found is that our performance is dramatically impacted by how well our subdivided shapes relate to sectors. If these subdivided shapes are triangles from an icosahedral subdivision (or the right triangles formed by splitting ones that cross the date line), the triangle area as compared to its bounding sector area is sufficiently small that performance is seriously impacted.

The experimental approach we describe here may prove to have similar problems in the polar regions. Our initial testing suggests that improvements related to the triangulation in the vicinity of the poles at least balances any new performance problems this may cause. Nevertheless, if we later discover this to be an issue, we can easily tune the approach by using a metric different from that discussed in section 3.2 for determining the ideal latitude cutoff point.

Aasgaard [1] describes a level of detail approach for tessellating the globe that produces identifiable vertical strips with boundaries along longitude lines. Within the strips, the tessellation – while fairly good from an aspect ratio perspective – has a fairly irregular topology. The computational requirements are also more significant than what is required for the approach we have developed.

Probably the most natural approach for GIS applications begins by generating quadrilaterals whose edges are parallel to lines of constant latitude and longitude (Fig. 4). These quadrilaterals can then be split along a diagonal to produce triangles. The initial release of World Wind used this method. While still an option, this paper describes the development and integration of a new experimental approach described below. An alternative approach was desired because, while the current one clearly works well over most of the globe and easily avoids triangles crossing the International Date Line, it clearly breaks down in the vicinity of the poles for all of the reasons mentioned above.

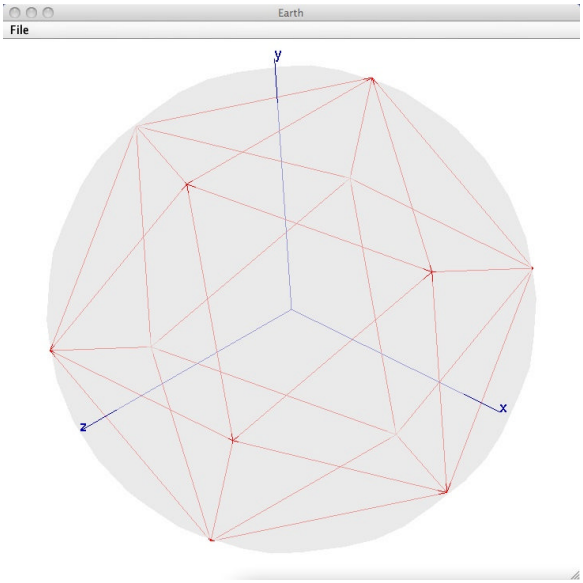

Fig. 3(a): An icosahedron inscribed in the ellipsoid.

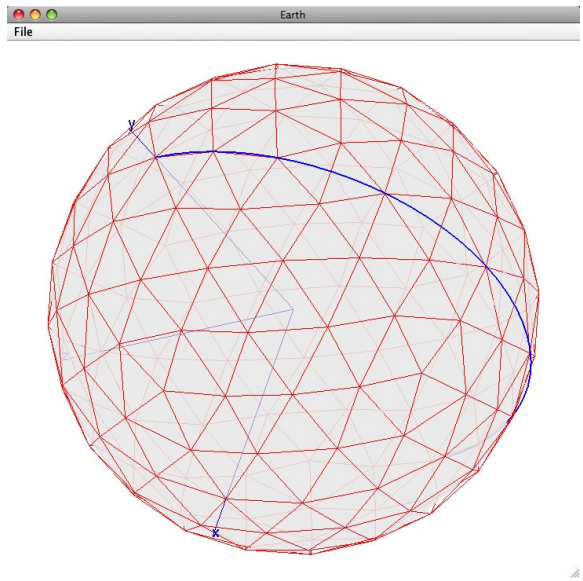

Fig. 3(b): After two refinement steps.

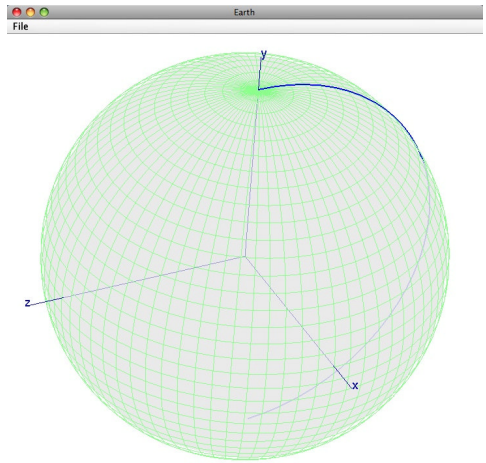

Fig. 4: A tessellation based on generating quadrilaterals aligned with latitude and longitude lines.

Mappings based on the use of a cube have also been explored. Fox and Joy [3], for example, developed a scheme based on inscribing a sphere inside of a cube. By recursively chamfering edges and vertices, the cube changes into a polyhedron whose shape approaches that of the sphere. Unfortunately this algorithm produces a collection of *n*-sided polygons. In fact, different polygons at a given level of recursion will exhibit different values of *n*. Subsequent triangulation or other scheme to handle elevation interpolation as described above would therefore be required.

3. SPECIALIZED TESSELLATION IN POLAR REGIONS

While the method of Fox and Joy does not seem to be well-suited for our environment, the experimental approach we have developed starts in a similar way. We begin by mapping the six faces of a cube onto the ellipsoidal globe (Fig. 5(a)). Notice that one edge of the cube is placed along the International Date Line. The four lateral faces can then be subdivided in a rectangular network. This is done in the straightforward way by subdividing along lines of constant latitude and longitude. To avoid the degeneracies that can arise in the polar region, we use a different, but compatible subdivision scheme for the top and bottom faces of the cube. By "compatible", we mean the tessellation vertices created along the shared edges will be identical at a given level of refinement.

The first level of subdivision in the polar region is defined so as to (i) preserve the property that no triangle will cross the International Date Line, and (ii) establish four polar sectors that can be subdivided an arbitrary number of times without leading to degenerate shapes.

Notice in Fig. 5(b) that four regions that appear to be triangular in shape are created in this initial subdivision. In fact, the four regions are four-sided; red dots mark a vertex between a pair of adjacent edges of constant latitude.

Second and subsequent subdivisions in the polar regions are accomplished by subdividing each of the four edges at its midpoint and defining new sectors by connecting the new vertices. (Compare Fig. 5(b) to Fig. 5(c).) During repeated subdivisions, the triangular-looking quadrilateral regions get pushed away from the poles towards the midpoints of the original cube edges (Fig. 5(d)).

The new edges that get created in the polar region clearly have neither constant latitude nor longitude. In order to produce an ideal tessellation with convex regions at each step, these edges must be shortest path edges along the ellipsoid. That is, they must be great ellipses.

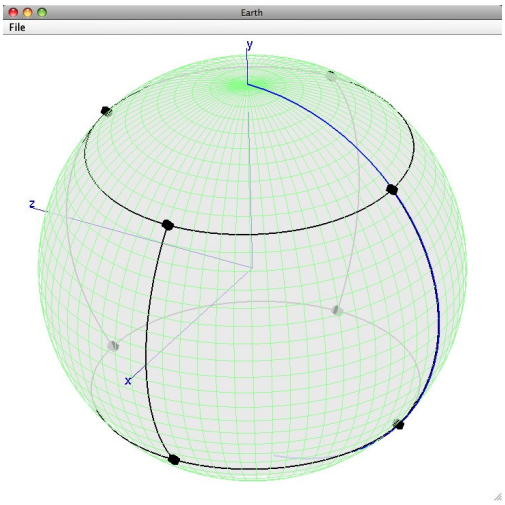

Fig. 5(a): Mapping a cube onto the globe.

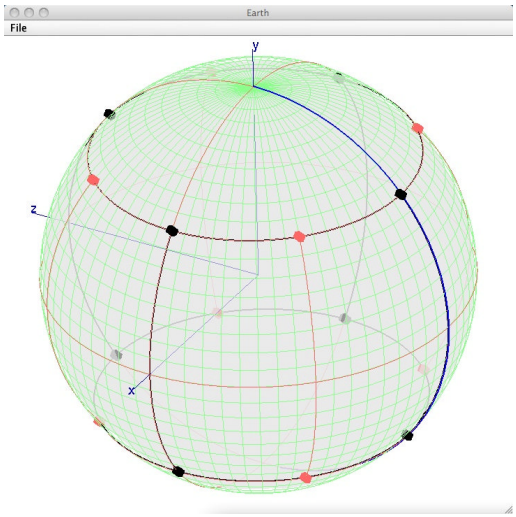

Fig. 5(b): First level of subdivision.

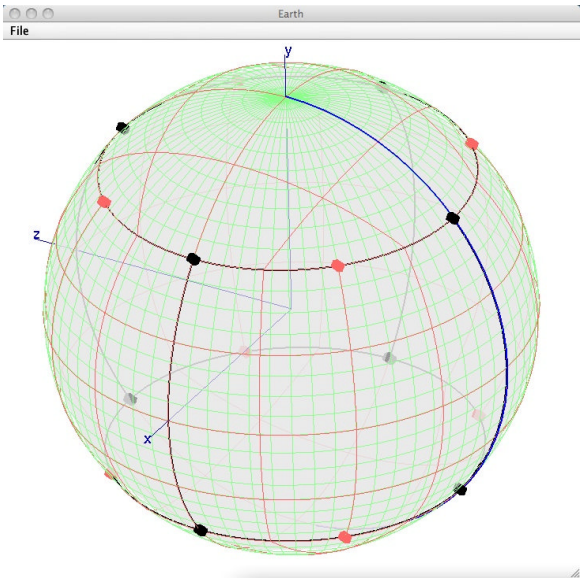

Fig. 5(c): Second level of subdivision

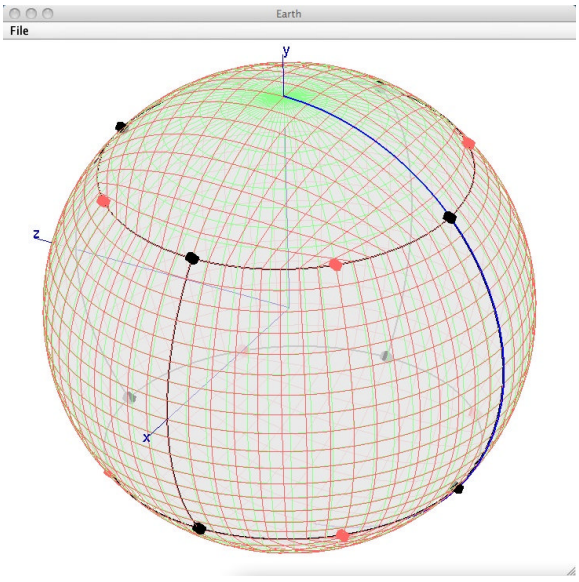

Fig. 5(d): Fourth level of subdivision

3.1 Triangle Generation

Rather than subdivide quadrilateral sectors down to the size of displayable triangles, we terminate the recursion somewhat earlier based on the current field of view. Each resulting quadrilateral intersecting the current view frustum is then triangulated by generating an *nxn* array of vertices inside of it. This strategy represents a tunable compromise between lots of recursion on the one hand versus the number of triangles generated inside a quadrilateral on the other. The generated *nxn* array of vertices is then used to define triangle strips. (See Fig. 6.)

To avoid introducing cracks in the tessellated surface, care must be taken in the generation of triangle coordinates. Fig. 6 illustrates a region of the globe where the polar region meets the lateral sides. We see portions of two of the top edges of the original cube appearing as a circular line of constant latitude ($\phi=\phi_0$) passing just south of the Great Lakes. Let us refer to the triangular-appearing quadrilateral in the middle just above it as *Q*, and note that *Q* is adjacent to two other polar region quadrilaterals on its left and right. The triangle vertices generated along latitude $\phi=\phi_0$ are computed

using constant latitude ($\phi=\phi_0$) in keeping with the triangles south of it. By contrast, as we move from the tip of *Q* and proceed south, a different vertex interpolation strategy is required. We march in parallel down the two edges of *Q* that meet at its northern tip. To interpolate from the left to the right edge, it is more consistent with the global tessellation strategy to generate points along a great ellipse arc connecting corresponding points. To avoid a discontinuity arising from an abrupt and discrete change from great ellipse interpolation to line of constant latitude interpolation, we compute points throughout *Q* by linearly interpolating between a great ellipse point, GE, and a line of constant latitude point, CL, as: $V = (1-t)GE + tCL$, where $t=0$ at the top of *Q*; $t=1$ along its bottom edges. This strategy is used not only for the triangular-appearing quadrilaterals like *Q*, but for any polar region quadrilateral that has an edge of constant latitude $\phi=\phi_0$. In fact, the need is even greater for those quadrilaterals, given how they meet other quadrilaterals along their other bounding edges.

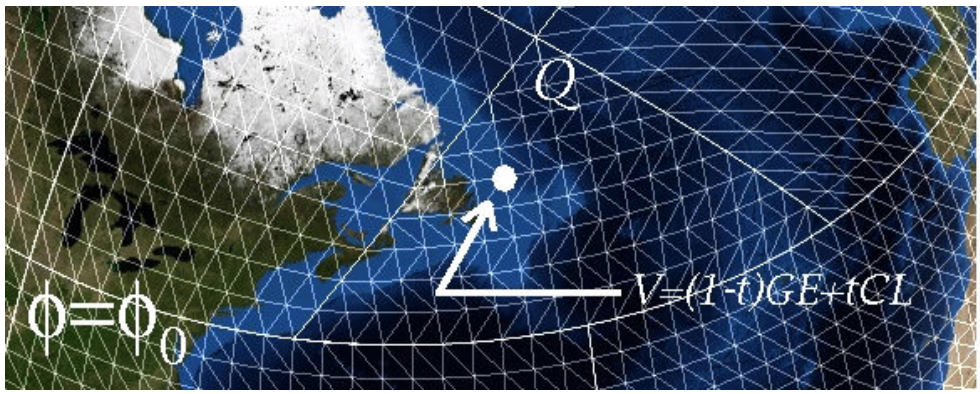

Fig. 6: An area where the polar region meets the lateral sides of the globe.

Textures imported into World Wind are georeferenced in terms of latitude-longitude coordinates. However the World Wind architecture keeps texture space strictly decoupled from geometric space, hence applying texture in the polar regions to triangles that have arbitrary relationships to the latitude-longitude grid is not a problem.

3.2 Determining a Suitable Latitude Cutoff Point

At least two competing criteria could be used to determine an optimal latitude cutoff point, $\phi=\phi_0$. On the one hand, we may wish to maximize the number of quadrilaterals bounded exclusively by lines of constant latitude or longitude. This would suggest we use relatively large values of ϕ_0, perhaps as large as 60 or 70 degrees. An alternative criterion would be to try to keep all quadrilaterals at a given level of subdivision roughly the same size. It is straightforward to determine the exact latitude at which the top edges of our cube will be mapped to elliptical arcs on our idealized ellipsoid that have the same length as that of a vertical cube edge mapped onto the ellipsoid. This latitude is approximately 40 degrees, and we have found that value to work well. In fact, all images in this paper were generated with $\phi_0=40$.

3.3 Other Issues

One of the significant issues that must be addressed when tessellating objects as described here is to avoid the generation of cracks between polygonal pieces. Adjacent sectors are frequently subdivided at different levels, hence cracks will appear if no remedy is applied. The situation is compounded by the fact that the elevation model resolution is constantly changing as we zoom and pan. Several schemes have been developed for World Wind and other tessellation-based rendering systems. Generation of "skirt" triangles at edges of recursively subdivided sectors can be used to effectively hide such cracks. This is the primary approach currently employed in World Wind. "Patch" triangles have also been employed in some applications. These triangles are generated by directly connecting vertices on adjacent regions of the surface that have been tessellated to different resolutions. A patching

algorithm would have to run at frame rates in World Wind since both the tessellation and the elevation model resolution change from one frame to the next; this capability has not yet been incorporated in World Wind.

To maximize rendering speed and performance, we cache many objects such as tessellation vertices and polygon connectivities so that they need not be recomputed each frame. However, for all the reasons cited above related to interactive zooming and panning, several cache validation strategies have been developed to ensure that relevant objects get updated as needed from one frame to the next.

4. OTHER USES OF THE TESSELLATION

Application programmers can place their own 3D geometry on, inside, or outside of the globe by querying the dimensions of the globe, using latitude-longitude coordinates, or some other equally straightforward scheme. On the other hand, the programmer sometimes needs to identify precise regions of the surface of the globe in order to overlay some color or pattern directly on the surface. In these cases, an implicit requirement is that the color or texture always lies exactly on the surface, regardless of zoom level. This in turn requires that the corresponding OpenGL primitives be redefined at least as often as the local surface tessellation changes. This presents a potential conflict between the desire to shield as much of the tessellation logic and data structures from the API as possible on the one hand and the need for the API to provide access to the tessellated triangles on the other.

Our current approach is to provide a simple facility to programmers that allows them to extract triangles and portions thereof that lie inside a given shape at the current zoom level. A shape outline is defined using latitude-longitude vertex coordinates. If the shape is potentially concave, the system will initially decompose it into triangles using the OpenGL GLU tessellation facility. In any event, the one or more convex outlines are used to define a cone-like prism whose vertex is the center of the globe and whose cross-section is the provided shape. A Sutherland-Hodgman polygon clipping algorithm is run on the tessellated triangles in the region of the defined shape, and triangles inside of or trimmed to this cone-like prism are returned to the caller. A caching scheme that is sensitive to the changing tessellation allows programmers to minimize these requests. Fig. 7(a) shows a hexagon defined over a portion of the Rocky Mountains; Fig. 7(b) shows a zoomed in view along an edge. Note how the shape – defined strictly by 6 latitude-longitude pairs – follows the terrain.

Since the shapes are defined as sets of vertices defined in latitude-longitude space, we need to define the nature of the edge between a pair of vertices. If a given pair of vertices has a common latitude (or longitude), then the intuitive and expected result is a corresponding edge of constant latitude (or longitude). However if they have neither, the expected (or even an ideal) result is not as obvious. Before discussing the approach we adopted, let us review some alternatives. Points along an edge could be defined using linear interpolation in latitude-longitude space. While this is easy to implement, the resulting edges are non-planar curves embedded on the sphere. In fact, the shape of two edges between two different pairs of vertices with the same (delta-latitude, delta-longitude) will be different if they are centered at different latitudes. Given that shapes defined in this way are frequently dragged interactively across the globe, this could be problematic because it means the shape will change as it is dragged (assuming dragging is performed by moving just the vertices by a constant distance across the globe, then redrawing the shape).

Other interpolation strategies along an edge between two latitude-longitude vertices are available. Rhumb lines (curves that cross longitude lines at a fixed angle) are important in navigation, but are perhaps not as useful in this context. We mentioned great ellipse edges earlier since they define the shortest path between two points on the ellipsoid. Since they also produce a shape dependent only on the (delta-latitude, delta-longitude) of the constituent edges, the shape of an object will not change as it is dragged around. This is what we do for edges that have neither constant latitude nor constant longitude.

In summary, then, the shapes described here use an adaptive strategy for determining edge types. An edge is generated as a constant latitude or longitude edge, if possible. Otherwise it is generated as a great ellipse edge.

Fig. 7(a): A hexagon defined over Rocky Mountains.

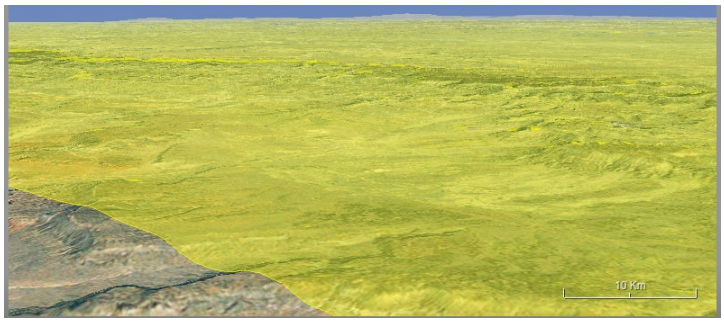

Fig. 7(b): After zooming in on a portion of an edge.

The collection of shapes that can be defined in this way either have polygonal boundaries or an elliptical boundary. Several convenience subclasses are available for the former including one that creates a regular *n*-gon inscribed in a circle or ellipse, a couple that define rectangular cross sections, and a general *n*-vertex polygon. The GLU tessellator is only used when an instance of a general *n*-vertex polygon is created.

Shapes can also be created from an actual elliptical boundary. In this case, a modified Sutherland-Hodgman algorithm that operates using a single ellipsoidal cylinder halfspace is used to trim and extract the desired portion of the tessellation.

Once defined, shapes can be interactively dragged over the surface of the globe without changing shape. The dragging algorithm determines a distance along a great ellipse edge by which each vertex is to be moved.

5. CONCLUSIONS

Ellipsoidal tessellations and related operations in a GIS environment involve an interesting interplay between traditional geometry modeling and physical modeling and geography. The latter brings with it geometry data typically derived from a variety of physical measuring mechanisms, typically post-processed using various interpolation and resampling schemes guided by assumptions related to the physical body. Especially in a distributed web-based multi-resolution application environment requiring a wide variety of physical data captured at multiple resolutions, the challenge is to effectively deal with pure geometric models at multiple resolutions while integrating auxiliary geometric data derived from the physical measurements and models, also captured at multiple (and generally different) resolutions.

6. REFERENCES

[1] Aasgaard, R.: Projecting a Regular Grid onto a Sphere or Ellipsoid, Symposium on Geospatial Theory, Processing and Applications, Ottawa, 2002.
[2] Arc GIS Explorer, http://www.esri.com, ESRI.
[3] Fox, D. E.; Joy, K. I.: On Polyhedral Approximations to a Sphere, Proceedings of Computer Graphics International, Hannover, Germany, June 1998, 426-432.
[4] Java OpenGL, http://jogl.dev.java.net, java.net.
[5] Google Earth, http://earth.google.com, Google.
[6] Teanby, N. A.: An icosahedron-based method for even binning of globally distributed remote sensing data, Computers & Geosciences, 32, 2006, 1442-1450.
[7] Virtual Earth, http://www.microsoft.com/virtualearth, Microsoft.
[8] Wernecke, J.: The KML Handbook: Geographic Visualization for the Web, Addison-Wesley, Upper Saddle River, NJ, 2009.
[9] World Wind, http://worldwind.arc.nasa.gov/java, National Aeronautics and Space Administration (NASA).