Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2003)
G.-P. Bonneau, S. Hahmann, C. D. Hansen (Editors)

Rendering Vector Data over Global, Multi-resolution 3D Terrain

Zachary Wartell[1], Eunjung Kang[1], Tony Wasilewski[1], William Ribarsky[1], Nickolas Faust[1]

[1] Graphics, Visualization and Usability Center, Georgia Institute of Technology, Atlanta, Georgia, USA

Abstract

Modern desktop PCs are capable of taking 2D Geographic Information System (GIS) applications into the realm of interactive 3D virtual worlds. In prior work we developed and presented graphics algorithms and data management methods for interactive viewing of a 3D global terrain system for desktop and virtual reality systems. In this paper we present a key data structure and associated render-time algorithm for the combined display of multi-resolution 3D terrain and traditional GIS polyline vector data. Such vector data is traditionally used for representing geographic entities such as political borders, roads, rivers and cadastral information.

Categories and Subject Descriptors (according to ACM CCS): 1.3.3 [Computer Graphics]: Display Algorithms

1. Introduction

GIS and 3D visualization are intimately related; one can empower and enlarge the other. This was our premise several years ago when we started a project to merge interactive visualization with GIS. This merger enlarges and empowers GIS because it offers a fully 3D GIS that can be interactively explored and displayed, which was our goal from the beginning. The merger enlarges and empowers interactive visualization because it gives visualization concrete and meaningful applications. Ultimately our work resulted in VGIS [2][3][6][7][8], a global geospatial system with scalable, multiresolution data organizations that permit one to explore comprehensive terrain (elevation and imagery), urban areas and 3D objects, orthorectified maps, 3D dynamic weather, and other geospatial data within the same framework. Among the GIS features in VGIS are its efficiently queryable geospatial databases and its ability to embed GIS annotations [2] (e.g., names, contents, and maps of buildings) in its structure for access through the visual interface. The queryable database is of significant use, for example, in providing terrain data for mesoscale weather models or determining the extent of flooding in an area. The GIS capabilities of the system would be significantly extended if vector data could be efficiently stored, displayed, and queried. Vector data include type, shape, and display properties for roads, state or county boundaries, rivers, property lines, and countless other features that enable GIS systems. Vector data enable, for example, the proper drawing of roads at all resolutions and the visual decorations that distinguish between road types. Since the vector data are also queryable, they permit quantitative measurements (e.g., road lengths between selected points) and attachment of identifying information (e.g., road names and characteristics). The addition of traditional vector data to the VGIS multi-resolution rendering capability and the methods required to do this are the subject of this paper.

Traditional Geographical Information Systems (GIS) display and compute with 2D geometric representations of geographic data. These data are typically organized as either raster data or vector data (perhaps more precisely called "coordinate data") [10]. Raster data are analogous to a bitmap or a regular 2d array where each array element contains a data value for a corresponding rectangular cell in the 2D plane. Vector data represent geometry as lists of coordinates that define points, lines and polygons. A key issue in both the computer graphics community and the GIS and computerized cartography community is eliminating unnecessary or unwanted geometric detail from the displayed image. This is done for both perceptual and computational efficiency. The computer graphics literature refers to this process as level-of-detail (LOD) management. In the GIS and cartography literature the term is "geometric simplification" [10]. GIS geometric simplification is one aspect of the more general methods of "generalization", which are methods for visually representing a given geographic entity or set of entities using different visuals when displaying the entities at different map scales.

This paper presents a key data structure, *the triangle clipping DAG* (direct-acyclic graph) and associated algorithms for overlaying traditional 2D polyline GIS data on a global, 3D terrain visualization system. Displaying 2D polyline data on top of global 3D terrain becomes challenging in our global system for several reasons. First, the terrain's 3D geometry data, image data and the 2D polyline data are too large to fit into primary memory. This requires dynamic paging of all three data types

© The Eurographics Association 2003.

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

based on the current 3D view of the database. For a number of reasons to be discussed shortly, we believe that 2D polyline data should be treated independently from the image data and therefore should be rendered as separate geometry by the graphics pipeline. This presents a challenge because modern terrain LOD algorithms render a 3D mesh whose constituent triangles are changing at nearly every frame. In order for the polyline data to appear overlayed on the 3D mesh, the rendered polyline geometry must therefore also change at each frame. In this paper, we describe the many challenges in developing a complete and scaleable algorithm for the combined display of polyline data and global 3D terrain. This paper then presents a data structure and algorithm optimized for the render-time component of a complete algorithm. We present performance results of the implemented algorithm. We also outline our current efforts to develop efficient secondary storage formats and to better balance the computation/memory trade-offs between the preprocessing-time and data-page-time algorithm components.

2. Prior Work

The combined display of traditional GIS vector data and global, multiresolution 3D terrain builds upon work published in both the computer graphics and GIS communities. This section briefly reviews the basics of terrain rendering, line simplification, and the complexities of combining them.

2.1. Terrain LOD Algorithms

LOD algorithms for 3D terrain rendering are an active area of research [5][7][9]. In this paper, we use the classic algorithm of Lindstrom et al. [7] for terrain rendering. Briefly, the Lindstrom terrain mesh algorithm partitions the height-field via a quadtree structure. The quadtree elements are quadnodes. Each quadnode has a block of 256x256 triangles (129x129 vertices). Later the algorithm was extended to global data through the use of a set of 32 linked quadtrees to represent the earth's spheroid [3].

Generally each triangle in the triangle block can be coalesced with a partner triangle into a single lower resolution triangle and conversely lower resolution triangles can be split into two higher resolution triangles. This defines a finite binary relation between triangles which induces two trees on the quadnode's mesh. This binary relation is called the triangle-split relation and the trees are the triangle-split trees. The top two nodes in the tree pair correspond to the lowest resolution triangle pair that could be rendered for the quadnode's mesh. Triangles deeper in the tree are of higher resolution. Triangles shallower in the tree are of lower resolution. Figure 2A illustrates the highest resolution mesh using, for illustration, a 9x9 vertex block. Figure 4 illustrates a small subset of the possible triangles the LOD triangulation algorithm may produce. This type of triangulation is variously referred to as 4-k meshes, right-triangulated meshes or restricted quadtree triangulations (RQT). The basic concept of our paper's algorithm assumes such a RQT triangulation. Certain details of our tc-DAG algorithm are specific to the Lindstrom mesh algorithm [4] and would have to be changed if one wanted to use recent generalized approaches such as that of Lindstrom and Pascucci [9].

2.2. Line Simplification Algorithms

Quite likely the notion of geometric simplification methods originated with cartographers as they represented a physical world of infinite detail on a map that can only usefully present a finite amount of detail. Cartographic simplification is one aspect of the more general concept of cartographic generalization that aims to address the general problem of complexity and detail through operations such as simplification, smoothing, aggregation, exaggeration and displacement [10]. Cartographers apply these techniques because simple photographic reduction of a map (i.e. image scaling), typically yields highly cluttered and incomprehensible maps. In this paper, however, we only focus on GIS line simplification, which simplifies geometric detail.

2D line simplification algorithms in digital cartography and GIS date back to the Douglass-Peucker algorithm presented in 1973 [10,p129]. In addition to generalization's main goal of representing important details clearly, when using computer systems we also want to avoid the computational cost of rendering unnecessary pixels. Chapter 10 of Longley et al. [11] reviews the GIS literature regarding line generalization through the mid-1990's. Line simplification and generalization continue to be an area of research [1]. However, to our knowledge this body of work continues to focus only on 2D polylines since GIS applications are traditionally 2D graphics applications.

2.3. Complications of Polyline-Mesh Combination

2.3.1. Texture Versus Geometry

There are two general approaches to rendering polyline vector data on a 3D mesh. One option is to convert the polyline data to a texture image layer and combine this polyline image layer with the primary terrain image layer (e.g. from a satellite/aerial photograph). The second option is to render the polyline data as separate 3D geometric primitives. Both techniques present a number of complications. This paper pursues the latter technique only, but the following paragraphs point out some of the complications of both of these approaches.

A naïve polyline-as-texture solution is to rasterize the polylines into the primary terrain texture image at the image's highest resolution and then render the terrain in the standard way using mipmaps or other suitable filtering. However, this is a poor solution because when zoomed out on the 3D terrain, the user will find much of the polyline vector information filtered away especially if single pixel lines were used for the rasterized vector data. This could cause state borders to be nearly completely filtered away when zoomed out to view an entire country. Additionally, it would be undesirable to zoom into a region of low primary image resolution and then to see not only the individual texels of the primary image data but also the individual texels of the texture-rasterized polyline vectors. The basic GIS semantics of a polyline line segment is that the represented geographic entity is defined by a line between two points in the real-valued Cartesian plane. The abstract line's accuracy is independent of the resolution of the primary imagery and we would not want to limit its visual representation's accuracy to the texel resolution of the primary imagery. For example, imagine the system displays a home owner's property line on top of the primary imagery. This

© The Eurographics Association 2003.

imagery might be either 50 M satellite imagery or 1 cm aerial imagery. In either case, the accuracy of the property line information does not vary. If we simply rasterized the property line into the primary terrain image layer under these two different conditions, each condition's visual results would convey very different notions of property line accuracy due to the large variation in texel resolutions of the texture-rasterized property line. Further, this native polyline-as-geometry approach does not allow the flexibility in polyline rendering expected in a GIS. These systems provide interactive enabling and disabling of the display of different subsets of polyline data and interactive adjustment of line styles such as line color, width and stipple patterns in order to distinguish and highlight different geographic data.

A 3D GIS needs a solution that can control the polylines' screen image independently from the primary texture. This indicates using either the polyline-as-3D-geometry solution or an adaptive polyline-as-texture solution. An adaptive polyline-as-texture solution would need to treat the primary texture imagery and the polyline rasterized texture as separate image layers of differing resolutions and would store the polyline in the original 2D geodetic coordinates. The polylines rendered via textures should have the same flexibility as if the polylines were rendered as 3D geometric OpenGL primitives (GL_LINES) allowing line color, stipple and width to be varied. The polyline-as-texture approach requires on-the-fly generation of the polyline-texture image because a-priori there is no limit on the zoom levels a user might choose and hence the maximum polyline-texture resolution needed avoid polyline texel enlargement is unknown. If we cache the textures, a cached invalidation policy is needed. Generating polyline-textures on the fly requires polyline simplification. We aim to explore an image-based polyline simplification scheme but this requires information on the polyline's structure in 3D space (i.e. how it exists as a 3D line on the terrain). This 3D information is not available in the polyline-as-texture approach because this approach explicitly avoids computing the polyline's 3D structure. For these reasons, we made a strategic decision to explore the polyline-as-3D-geometry approach first. Ultimately a comparison of two complete, scalable, and global implementations of both techniques should be performed.

2.3.2. Other Polyline-as-geometry Work

IMAGINE by Erdas is a traditional 2D GIS package that has added a separate 3D terrain visualization module called "IMAGINE VirtualGIS" [4]. This module is a simplified version on the algorithm of Lindstrom et al. [7]. The module, does not provide a global terrain database and the VirtualGIS LOD algorithm does block-based LOD but does not perform lower level triangle LOD. These facts can be easily verified by running VirtualGIS in wireframe mode. VirtualGIS can display polyline vector data. The method appears to be a polyine-as-geometry approach. The fact that only block-based LOD is performed, however, simplifies the polyline-as-geometry problem because once a given block is chosen for rendering the set of rendered triangles within that block never changes. In contrast, in the more sophisticated RQT triangulation algorithms, the set of rendered triangles change in a much more arbitrary way from frame to frame.

3. Methods

LOD algorithms with dynamic paging, split computation over three phases: preprocessing, data-load-time and render-time. A preprocessing step computes LOD information such as edge-collapse tolerance values, performs re-meshing and coordinate system transforms and constructs some LOD data structure, typically a tree or DAG. These data are stored in secondary storage. During interactive viewing there is data-load-time computation and a render-time computation. The data-load-time computation loads and unpacks the data from disk into primary memory. A typical example is transforming geodetic height field data stored in secondary memory into easily rendered Cartesian coordinates in primary memory. Finally, at render-time for each frame or iteration of the LOD algorithm, the primary memory data structures are examined to determine what geometry to send to the graphics pipeline.

The triangle clipping DAG structure and associated algorithms are a key element in combining traditional 2D GIS line simplification methods with 3D terrain LOD rendering. The presented algorithms provide an efficient solution to many of the render-time issues. In the following sections, we discuss these in detail and point out where further work is needed regarding aspects of the preprocessing algorithm and regarding the computation/storage balance between the data-load-time and preprocessing components.

3.1. Data Structures

```
struct quadnode
   {
   /* terrain and object data members */
          ……
   /* polyline level of detail */
   polyline_LOD_t polylineLODs [];
   }
struct polyline_LOD_t
   {
   /* graphical attributes */
          ……
   /* quad clippings */
   quad_clipping_t quadClippings[];
   }
struct quad_clipping_t
   {
   triangle_clipping_t triangleClippings[];
   }
struct triangle_clipping_t
   {
   point3D_t points[];
   triangleID_t ID;
   vertex_index_t decisionVertex;
   triangle_clipping_index_t  nextTriangle[2];
   triangle_clipping_index_t  parent;
   triangle_clipping_index_t  finest_exit_descendent;
   }
```

Figure 1: Pseudo-code for Data Structures

© The Eurographics Association 2003.

In its global form [3], the Lindstrom terrain algorithm partitions the spheroidal height-field into a set of 32 quadtrees. The quadtree elements are quadnodes. Each quadnode has block of 128x128 triangle pairs. For each quadnode, Q, the basic tc-DAG algorithm adds a dynamic array of *polyline_LOD_t* to the quadnode structure to represent the polylines crossing Q. A *polyline_LOD_t* (Figure 1) contains information on the polyline's visual attributes such as width, color and line stipple and a dynamic array of *quad_clipping_t* structures. A *quad_clipping_t* represents a single section of a polyline that has one entrance and one exit from quad Q. Since a polyline may enter and exit Q multiple times, we have potentially multiple quad clippings per *polyline_LOD_t*. In Figure 2, we show only an 8x8 triangle-pair block (9x9 vertices) for simplicity. Figure 2A shows the original polyline. Figure 2B shows the two quad clippings for the polyline traversing Q. A *quad_clipping_t* contains a dynamic array of *triangle_clipping_t* structures. A *triangle_clipping_t* corresponds to a single triangle in any legal RQT built on Q's vertices and represents a single section of the polyline that has one entrance and one exit from the triangle clipping. The entrance edge is the triangle edge the polyline enters. The exit edge is the triangle edge the polyline exits. (Possibly either is non-existent if the polyline starts/stops inside the triangle.) Figure 2B illustrates the two quad clippings broken up into individual line segments per triangle clipping. Successive triangle clippings are in alternating colors. Note in Figure 2B we only illustrate the triangle clippings for triangles in the highest resolution RQT mesh for this quad. In general, a triangle clipping can come from any legal triangle that the RQT LOD algorithm might build from the quad's triangle vertices. Figure 4 illustrates all possible RQT triangle clippings for one quad clipping crossing a 9x9 block. Since a polyline may enter and exit a given RQT triangle multiple times, a single polyline may have multiple triangle clipping's covering the same RQT triangle. (Note, none of the figures illustrate this possibility.)

A *triangle_clipping_t* contains geometric, topological, and identification information (Figure 1). The geometric component is an array of 3D point coordinates representing the successive points of the polyline as projected onto the plane of that triangle clipping. A *triangle_clipping_t* then has a size 2 array of indices to the "next" triangle clipping. The data member is called *nextTriangle[2].* This indexes the triangle clipping that the polyline enters after exiting the current triangle clipping. It is easy to show that because of the RQT triangulation structure a triangle clipping can have either 0, 1 or 2 next triangle clippings. The LOD of a *nextTriangle* will either be equal to, one level above or one level below the LOD of the current triangle clipping. The *nextTriangle* member defines a finite N-to-N binary relation on the triangle clippings within the context of a given quad clipping. This induces the primary directed-acyclic-graph on the triangle clippings where the nodes are triangle clippings with in-degree of 0, 1 or 2 and out degree of 0,1 or 2. A sentinel value is assigned to a *nextTriangle* element to indicate the absence of a next triangle. The *nextTriangle* is illustrated in Figure 4 with black arrows. (We don't show these arrays for the last two columns since the image grows too dense to make sense of). Additionally, each triangle clipping structure, T, contains an index to the triangle clipping at one LOD shallower in the triangle-split tree that contains T. This data member is called *parent*. A member *finest_exit_descendent* indexes the triangle clipping's descendent triangle clipping (in the triangle-split tree) of highest resolution whose exiting edge is embedded in the T's exiting edge or that contains the polyline's end point.

Figure 2: Polyline over a quad node with just a 9x9 block of vertices in RQT triangulation.

3.2. Preprocessing Algorithm

The preprocessing algorithm builds all *polyline_LOD_t* structures and writes them to secondary storage. The goal is to find every valid RQT triangle over a quad's vertex block that is intersected by a polyline quad clipping. For each such RQT triangle there is at least one and perhaps many triangle clippings depending on the number of times the quad clipping enters and leaves the triangle. The triangle clippings must be strung together using *triangle_clipping_t.nextTriangle* in the order in which the quad clipping visits the triangles. To do this, we trace the quad clipping through triangle-pair cells (tp-cells) at the highest resolution RQT. TP-cells are the latitude-longitude squares defined by pairs of triangles sharing a

© The Eurographics Association 2003.

diagonal, hypotenuse edge. Figure 2A a single tp-cell is circled in the lower-left corner. The 9x9 vertex array has 8x8 tp-cells.

After tracing the quad clipping through the highest-resolution triangles, we iterate over these successive triangle clippings and at each clipping climb up the triangle-split graph to find the ancestors of the finest triangle clippings. As this is done, ancestor triangle clippings are added to the triangle clipping array. These further triangle clippings are added in order of highest resolution to lowest.

VGIS's quadtrees and height-field use a single, global, spheroidal coordinate system. Hence the original polyline coordinates are in latitude and longitude. These must be geodetically projected onto the mesh triangles. This coordinate system transformation and projection is far more floating point intensive than the trivial parallel projection that would suffice for non-planetary terrain systems [11]. As a partial example, in a non-global model, projection of point *a* to plane *P* is the intersection of the projector (a line) defined by point *(a.x,a.y,0)* and vector *(0,0,1)*. (Here we assume the height field rises along the z-axis). In contrast, just calculating the XYZ coordinate of geodetic projector's start point using *a*'s geodetic lat/lon coordinates involves 4 sin/cos evaluations and a square root. Intuitively and empirically, performing these geodetic projections at render-time is prohibitive. Instead we should perform the geodetic projection at either data-load-time or preprocessing time. Presently, we perform the projection at preprocessing time and hence both the primary and secondary memory *triangle_clipping_t* structure stores a list of Cartesian points.

Another complexity is the precise geometric tracing of spheroidal geodesics through tp-cells (Figure 2A). Spheroidal geodesics are the proper way to connect consecutive points in a lat/lon polyline. The naïve approach, which we currently use, is to approximate a mapping of the spheroid to the plane such that lat/lon squares on the spheroid map to lat/lon squares on the plane and that spheroidal geodesics map to straight lines in the plane. Under this approximation tracing geographic geodesics through the lat/lon grid only requires tracing a straight line through a Cartesian grid. However, it is well known in cartography that no such spheroid-to-plane (nor sphere-to-plane) mapping exists. The simple cylindrical projection maps a regular spherical lat/lon grid to a grid of squares in the plane but spherical geodesics map to complex curves. In the Polar Zenithal Gnomonic projection and Equatorial Zenithal Gnomonic projection spherical geodesics *do* map to lines but a regular lat/lon spherical grid maps to a grid of curves [14]. It is not yet clear which of these methods is best modified for tracing polylines over the spheroid.

At present, we directly write the resulting *polylineLOD_t* structures straight to secondary storage. This is not space efficient. The *polylineLOD_t* structures are optimized for render-time operations. Ultimately, a more compact, external format for secondary storage is needed. Details are discussed in 3.3.

3.2.1. Render-time: Polyline-Mesh Projection

At render time, the Lindstrom algorithm traverses the quad tree and selects which quad's vertex blocks should be rendered. When a block is reached and rendered, a triangle LOD pass activates and enables the necessary vertices within the block. Each vertex has a bit that says whether it was enabled or not. This information is used by the polyline renderer to determine what path through the triangle clipping DAG should be followed. As each triangle clipping node is reached its polyline vertices are rendered. All vertices for a given *quad_clipping_t* are rendered in a single glBegin/glEnd pair using GL_LINES. This maximizes efficiency for drawing long polylines.

We present the triangle clipping DAG traversal algorithm by first assuming a simplistic behavior of the RQT algorithm used for mesh LOD. Later we'll see that in practice implementations such as the Lindstrom algorithm behave in more complex ways and require more complex tc-DAG traversal mechanisms. A simplistic block-based RQT algorithm would behave such that when the RQT algorithm reaches a quad node, Q, the algorithm either examines all of the quad's vertex data for mesh rendering or examines none at all. Assume the RQT algorithm chooses to render Q's mesh, thus setting its mesh vertices' enabled bits. For each quad clipping QC of quad Q, we start with the first triangle clipping of highest resolution in QC's triangle clipping DAG. (By design, this is the first triangle clipping of QC's *triangleClippings* array). By following the *triangle_clipping_t.parent*, we visit all triangle-split ancestors until reaching a triangle clipping whose mesh vertices were enabled (i.e., flagged for rendering by the triangle LOD pass). This gives us the first triangle clipping to render so we render that clipping's polyline vertices. To determine which of the next triangle clippings to follow we test the enabled bit of the mesh vertex number N, where N is the *decisionVertex* structure member of the current triangle clipping. The following pseudo-code illustrates this algorithm. Note, a triangle clipping index of -1 implies there is no next triangle and *nextTriangle[0]* holds the next triangle when the triangle clippings node's out-degree is 1.

Algorithm 1:

```
render_polyline (polylineLOD_t PL, quad_mesh_t QM)
{
   for all quad clippings QL of PL
      {
      triangle_clipping_index_t TCI;
      TCI = find start triangle clipping in PL;
      while(TCI != NULL)
          {
          triangle_clipping_t TC;
          TC = QL.triangleClippings[TCI];

          /* draw all polyline points in a current
               triangle clipping */
          draw_points(TC);

         /* choose next triangle clipping */
         if(TC->nextTriangles[1] >=0)
           {/* nextTriangle[1] does exist, so
               choose between nextTriangle [0]
               and [1] */
           if(vertex_rendered(QM,tc->decisionVertex)
               /* decison vertex was rendered, so
```

© The Eurographics Association 2003.

```
          step to nextTriangle[0] */
         TCI =TC->nextTriangles[0];
       else
         /* decision vertex was NOT rendered,
           so step to nextTriangle[1] */
         TCI =TC->nextTriangles[1];
       }
     else
         /* nextTriangle[1] does NOT exist, so
            just examine nextTriangle [0] */
         TCI=TC->nextTriangles[0];
     }
   }
}
```

This simple algorithm is sufficient for a block-based RQT algorithm if and only if the mesh algorithm guarantees that in a given frame, it either considers the whole of a quadnode's vertex data block or does not consider it all. Unfortunately, we empirically observe that the Lindstrom algorithm [4] frequently traverses down to a quadnode, Q, and then borrows the needed vertex data from a sub-region of the vertex array of an ancestor quadnode, AQ. This borrowing appears to occur for several reasons. First, vertex data are paged in asynchronously compared to the building of the in-memory quadtree. So the block-level LOD algorithm may decide it needs to render quad Q's vertices, but Q's vertex data is not yet paged in. Hence, the mesh algorithm must temporarily borrow vertex data from some ancestor quad. Second, the Lindstrom algorithm associates a given LOD of both the vertex mesh and terrain imagery with a fixed quad Q. It is possible that the image data may be used from one quad level while the vertex data is from another. A third reason for vertex data borrowing is that the Lindstrom algorithm forces all quadnodes to have either 0 or 4 children which may lead to leaf quadnodes that have no vertex data associated with them.

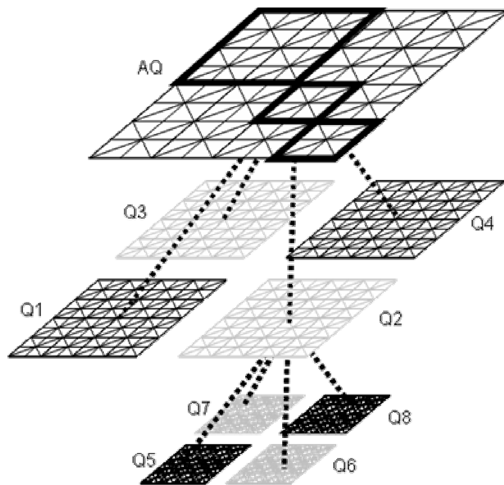

Figure 3: Example of Vertex Borrowing. Gray meshes are quadnodes with no loaded vertex data. Black meshes are quadnodes with loaded vertex data. The outlined sub-regions of AQ are the borrowed vertex data regions borrowed by descendant quads Q3, Q6 and Q7. All other regions (Q1, Q4, Q5, Q8) are rendered with their own data

Figure 3 gives an example of borrowing. Q3, Q6 and Q7 lack data and the quadtree traverser borrow's mesh data from AQ to render triangles for these regions.

Vertex data borrowing greatly complicates the implementation of Algorithm 1. With borrowing, when we render a quad clipping we render non-contiguous portions of the quad clipping as it wanders in and out of various disconnected borrowed sub-regions of AQ's (outlined in AQ in Figure 3). The triangle clipping DAG traversal for AQ cannot assume that each successively visited triangle clipping should be rendered. This incurs two performance penalties. First, we spend time visiting triangle clippings whose polyline point data must not be rendered (because none of the covering triangles in AQ were rendered in this frame). Second, every time the quad clipping re-enters a borrowed sub-region, the DAG traversal must determine which triangle clipping (1) has an edge covering the entrance point of the quad clipping to the borrowed sub-region and (2) has its corresponding RQT triangle rendered. This requires testing up to *covering_triangles* triangles for their enabled status, where *covering_triangles* is the number of RQT triangles that may cover a point projected on the mesh. This value is 2 * (*levels_of_detail_per_quad*-1) + 1 which is 15 for our Lindstrom algorithm implementation.

Our current solution for the borrowing problem is as follows. All polyline LOD data for a quad Q is paged in and out in unison with Q's vertex mesh data. Next, we distinguish three versions of Algorithm 1 each of which is executed in different circumstances. If no vertex borrowing takes place for rendering Q's mesh (Q1,Q4,Q5,Q8 in Figure 3), we render Q's mesh data and then execute Algorithm 1 to render Q's polyline data. Here the borrowed polyline data is drawn when the quad tree traversal visits node Q.

If borrowing does occur, we render polyline data with one of two other versions of Algorithm 1. The two borrowed-mesh versions use additional cached information. This cache stores intermediate computation results for borrowed mesh sub-regions in a structure called *sub_polylineLOD_t* (sub for "subset"). A quadnode Q will contain both a dynamic array of *polylineLOD_t* and a dynamic array of *sub_polylineLOD_t*. *sub_polylineLOD_t* contains a dynamic array of *sub_quad_clipping_t*. *sub_quad_clipping_t* references a subset of a quad clipping in the borrowed ancestor quad, AQ. In particular, each *sub_quad_clipping_t* references a section of one of AQ's quad clippings after it is clipped to Q's borrowed sub-region in AQ.

The first variant of Algorithm 1 is executed when Q must borrow mesh data from ancestor AQ and Q's cached sub polyline data is invalid. Instead of rendering Q's polyline when the quadtree traversal visits a quad Q (such as Q3,Q6,Q7 in Figure 3), we set some flags in AQ that indicate what mesh sub-region in AQ was borrowed by Q. Later, when the recursive quad tree traversal backtracks up the tree to AQ, it notes that descendents Q3, Q6, and Q7 of AQ borrowed AQ's mesh data. The quadtree traversal then calls the first variant of Algorithm 1 which renders AQ's polyline data but only for the borrowed sub-regions. In this case, the *tc*-DAG traversal must

© The Eurographics Association 2003.

pay the penalty for wandering in and out of various borrowed sub regions. However, the sub polyline cached structures are built and/or updated at this time.

Now that the sub polyline data is cached, when the quadtree traversal visits Q in the next frame, it executes the second variant of Algorithm 1. This variant uses the cached sub polyline data in Q to immediately grab the appropriate portions of AQ's quad clippings contained in Q. Note, in this case the borrowed polyline data is drawn when the quadtree traversal visits Q itself, instead of delaying the polyline rendering until backtracking to AQ.

The cached sub polyline structures take advantage of frame coherence in the terrain LOD thread. The creation/invalidation behavior of this cache is dependent on the pattern of mesh data borrowing. Empirically, the borrowing behavior cannot usefully be predicted a priori (i.e. in the mesh LOD preprocessing) since the behavior depends on the view point flight path. However, the pattern of borrowed data changes slowly enough under continuous view point movement to make this caching a performance advantage.

Here are several points regarding OpenGL rendering. First the tc-DAG traversal algorithm packs all of a quad clipping's points into a single OpenGL GL_LINES primitive. OpenGL vertex array calls and vertex caching extensions can also be used. To deal with z-buffer occlusion artifacts between the GL_LINES and the co-planar mesh, we use OpenGL polygon offset. One is tempted to use standard stencil buffer tricks and treat the GL_LINES like "decal" co-planar polygons [15,pg516]. However, for this to work for arbitrary view points, all of each mesh triangle's decals (here the embedded polylines) must be rendered immediately after each *individual* triangle. This, however, disallows using efficient triangle strips.

3.2.2. Render-time: Polyline LOD

The basic triangle clipping DAG structure provides a basis for adding polyline LOD. In this section, we explore some the possibilities and present some important observations.

Extending more recent 2D polyline LOD methods to 3D terrain systems raises many complications. For example, Oosterom's BLG (Binary-Line Generalization) Tree encodes results of the Douglass-Peucker algorithm into a tree structure as part of a preprocessing step [13]. The recursive algorithm takes the polyline end points and chooses the intermediate point which is the greatest distance from the line spanning the end points (Figure 5). This most distant, intermediate point is added to the tree along with its distance to the spanning line segment. This node represents an approximation to the original polyline where the approximation contains the end points and the chosen point. Next the algorithm is repeated, recursively treating each original endpoint and the newest chosen point as new endpoints. At render time, the render algorithm uses the resulting tree to quickly choose a subset of original vertices such that the error between the used approximation and the next finer approximation is below a given threshold. Oosterom incorporates this algorithm into an interactive 2D system. For a 2D system given a screen space error threshold, we only need to multiply by a constant scaling factor to map the error threshold value from screen space into the object space in which the BLG tree distance values were computed.

Unfortunately, this simple screen space to object space computation is invalid once a planar polyline is centrally projected onto the screen in a 3D application. In the 2D application the order in which points are inserted into the BLG tree is determined by the distances in object space. In the 2D application this order would not be changed by a similarity transform that maps object to screen space. In other words, the data structure is similarity-invariant. However, in the 3D realm, a planar polyline under different perspective projections would yield different relative error distances for the recursively added polyline points. This generally yields a different BLG tree structure for each possible 3D viewpoint. The BLG structure is not projective-invariant. So in the 3D environment the BLG tree can not be as easily and efficiently employed with a screen space error metric as in a 2D GIS application. A simple option is to use only an object space metric with some eye point to triangle clipping distance measure for determining the BLG rendering tolerance. Even better, however, we might try to apply similar methods used in the screen error metric RQT 3D mesh simplification strategies.

Note, the triangle clipping DAG structure is easily augmented with a BLG tree on a per triangle clipping basis. Clearly, adding a BLG tree to a triangle clipping is most advantageous when the number of polyline vertices per triangle clipping is high. Alternatively, one is tempted to try to construct a BLG across multiple triangle clippings. Call this alternative the BLG-tree-over-the-mesh approach. This second approach must perform the geodetic projection of every possible polyline that might result from the BLG tree evaluation. Each line segment of each possible BLG polyline would have to be traced through the RQT triangle mesh, split across triangle edges and geodetically projected onto the mesh. From our experience with earlier alternatives to the triangle clipping DAG, we suspect that a BLG-tree-over-the-mesh approach will require at least as much memory as the triangle clipping DAG's mesh-over-BLG-tree approach. Most importantly, we observe that in a finely tessellated RQT mesh, simplifying a polyline prior to projection may have little over all performance advantage in terms of the number of rendered lines. The reason is even if we simplify the polyline to a single line segment, this single line segment will be broken into many pieces when geodetically projected onto a fine mesh. *P0.quadClipping[1]* in Figure 2 is a prime example. On the other hand, if we have a coarse mesh, then the BLG-tree augmented tc-DAG should perform just as well assuming the original polyline is relatively smooth.

We are beginning to experiment with adding BLG tree's to triangle clipping.

3.3. Data Storage and Paging

The triangle clipping DAG and associated algorithms presented in this paper focus on the render-time component of a complete combined polyline-3D-mesh solution. At present our preprocessing step simply stores the triangle clipping DAG directly to secondary storage in binary format. Polyline vertex data is dynamically paged at the granularity of a quadnode's complete tc-DAG. The tc-DAG is paged in whenever the mesh

© The Eurographics Association 2003.

LOD algorithm retrieves the quad's vertex data. Hence the Lindstrom mesh algorithm gives us quadnode level data paging and view frustum culling automatically for polylines.

The triangle clipping DAG presented in section 3.1 is optimized for render-time computations. It can be quickly traversed and polyline vertex coordinates can be directly pushed into OpenGL GL_LINES primitives. A separate external format is still needed to reduce secondary storage and better balance the data-load-time CPU cost against its disk access cost. In this section, a discussion of the storage required by naïvely dumping the in-core tc-DAG to disk will motivate the design of a more compact secondary storage format.

A single polyline point projected onto a quad Q's mesh can be covered by *covering_triangles* triangles. Recall, 8 vertex LOD's per quad yields 15 covering triangles. Hence a quad clipping stores 15 projected copies of the original polyline vertices. This increases coordinate storage costs 15 times. Assume 12 bytes per coordinate (3 32-bit floats). Our Georgia County polylines on the ATL terrain (Figure 6) would use an estimated 240000*12*15 = 41 MB of storage for the coordinate data alone for a single low resolution quad mesh that covers the entire state. The empirical storage for this top level quad is 41.8 MB with 98% of the storage devoted to coordinate data.

To reduce this intra-quad redundancy, the external tc-DAG storage format should store the DAG topology and some form of the polyline vertex coordinates that is intermediate between the original lat/lon coordinates and the mesh projected Cartesian coordinates. A useful compromise is storing the geodetic projector for each lat/lon polyline vertex. These projectors are shared across triangle clippings of all LOD's and this choice pushes the trigonometry heavy geodetic transformation to preprocess-time leaving the simpler plane-projector intersection test to data-load-time.

For quick render-time access, the triangle clipping DAG stores all triangle clippings relevant to a quad in the quad's data structure. This allows quick access during render-time quad tree traversal. Generally this does not waste too much primary memory since the block-LOD algorithm generally avoids loading spatially overlapping quads' vertex data. (Recall, we page in and out the polyline structure in unison with the quad's mesh vertex data so by avoiding primary-memory vertex data duplication we avoid polyline vertex duplication as well). By dumping every quad's tc-DAG to secondary storage we store each quad's complete triangle clipping DAG. There is now inter-quad redundancy because parent and child quads share many of the same triangles. Except for the root quad, each quad only adds its 2 highest resolution triangulations to the global triangulation. A quad's other 13 (15-2) triangle LOD layers are already represented in ancestor quads. Total storage for the Georgia County preprocessed polyline data is 615MB with 68% of storage going to coordinate data. The obvious solution to the inter-quad redundancy is to only store triangle clippings for the 2 highest resolution triangulations for non-root quad nodes. We estimate that reducing both the inter-quad and intra-quad redundancies in a specialized external data format will reduce storage costs by ten fold. Further reductions are possible using delta encodings of the polyline coordinates.

4. Results

The results of running our tc-DAG algorithm for several data sets are shown in Figure 6. The first few images show Georgia county borders. The border data consists of 389 polylines using 240K points. The terrain database consists of 50 km elevation data for the world at large; 30 m elevation data for Georgia and 10 m data for downtown Atlanta. The software was run on a 1.5 GHz Pentium 4 with an NVidia Geforce 2 GTS graphics card. With the polylines disabled the system renders 82 triangle strips for the mesh with a total of 1820 vertices. The render thread runs at 60 FPS while the LOD thread runs at 2.0 FPS. When we enable the display of the polylines, another 407 OGL primitives are rendered corresponding to the 389 polylines. (In this scenario only a few polylines are evidently split across triangle clippings). The vertex count increases to 241605 vertices. The renderer thread FPS drops to 45 and the LOD FPS drops to 1.7. Depending on viewpoint for this dataset, the render and LOD FPS vary in a range of 15-20% from these values (assuming no loading of new data from disk). Of course we expect improved performance with a more recent graphics card. Figure 6 also contains zoomed in views that show the polyline following the 3D terrain. Figure 6 then illustrates other vector data--North and South Korea with borders outlined with polylines. We have also run the algorithm on polyline street data for Atlanta.

Polylines Enabled	Render FPS	LOD FPS	OGL Primitives	OGL Vertices
YES	45	1.7	489	241605
NO	60	2.0	82	1820

Table 1: Comparison of run-time statistics for top image in Figure 6.

5. Conclusions and Future Work

This paper presented the tc-DAG data structure and its render-time algorithm. These are key components for combining the display of polyline vector data and global 3D terrain. We showed and discussed results of our multi-resolution implementation and showed interactive rendering. We are developing a space-efficient external storage format and improving the balance between the preprocessing-time and data-paging-time computations. Additionally, the tc-DAG structure is ripe for the addition of a BLG-tree adaptation for multi-resolution polylines.

Acknowledgements

This work is supported by the Department of Defense's MURI program, administered by the Army Research Office.

References

1. Albert.H.J. Christensen, Line generalization by waterlining and medial-axis transformation. Successes and issues in an implementation of Perkal's Proposal, Cartographic Journal 37, no.1 (2000) p. 19-28.
2. Davis, Douglass, T.Y Jiang, William Ribarsky, and Nickolas Faust. "Intent, Perception, and Out-of-Core

© The Eurographics Association 2003.

Visualization Applied to Terrain," Report GIT-GVU-98-12, pp. 455-458, IEEE Visualization '98.

3. Nickolas Faust, William Ribarsky, T.Y. Jiang, and Tony Wasilewski, "Real-Time Global Data Model for the Digital Earth," Proceedings of the INTERNATIONAL CONFERENCE ON DISCRETE GLOBAL GRIDS (2000).

4. IMAGINE VirtualGIS V8.3, Tour Guide, ERDAS Inc., Atlanta GA, 1997.

5. Hugues Hoppe, Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. Proc. IEEE Visualization '98, pp. 35-42 (1998).

6. T.Y. Jiang, William Ribarsky, Tony Wasilewski, Nickolas Faust, Brendan Hannigan, and Mitchell Parry. Acquisition and Display of Real-Time Atmospheric Data on Terrain. Proceedings of Eurographics-IEEE Visualization Symposium, pp. 15-24, 2001.

7. Peter Lindstrom, David Koller, Larry Hodges, Bill Ribarsky, Nick Faust, and Gregory Turner, "Real-Time, Continuous Level of Detail Rendering of Height Fields,", Proceedings of SIGGRAPH '96, pp. 109-118 (1996).

8. Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Augusto Op den Bosch, and Nick Faust, "An Integrated Global GIS and Visual Simulation System".GVU Technical Report GIT-GVU-97-07, Graphics, Visualization and Usability Center, Georgia Institute of Technology, March 1997.

9. Peter Lindstrom and Valerio Pascucci, Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization., IEEE Transactions on Visualization and Computer Graphics, 8(3), pp. 239-254, July-September 2002.

10. Paul A. Longley, Michael F. Goodchild, David J. Maguire, David W. Rhind, Geographical Information Systems: Volume 1: Principle and Technical Issues, John Wiley & Sons, Inc., New York 1999.

11. Maling, D.H., Coordinate Systems and Map Projections, George Philop and Son Limited. London. 1973.

12. Robert Brainerd McMaster, Automated Line Generalization, Cartographica, 24/2: 74-11, 1997.

13. Peter van Oosterom. 'The Reactive-tree: A Storage Structure for a Seamless, Scaleless Geographic Database'. In proceedings Auto-Carto 10, Baltimore, Maryland, pages 393-407, March 1991.

14. Hugh S. Roblin, "Map Projections", Edward Arnold LTD. 1969.

15. Mason Woo, Jackie Neider and Tom Davis, OpenGL Program Guide, Addison-Wesley Developers Press, Reading, Massachusetts, 1997.

© The Eurographics Association 2003.

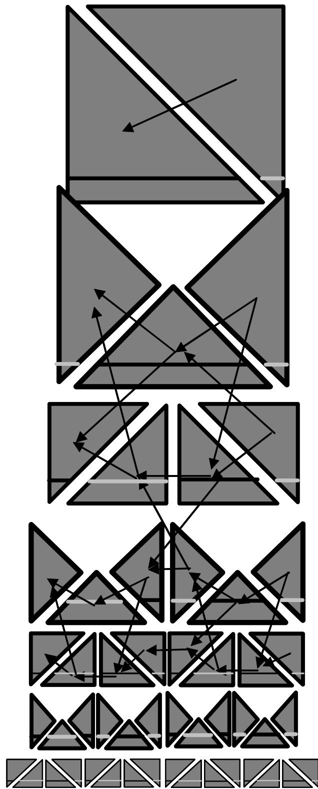

Figure 4: Triangle Clippings of a single segment quad clipping assuming a 9x9 vertex quad.

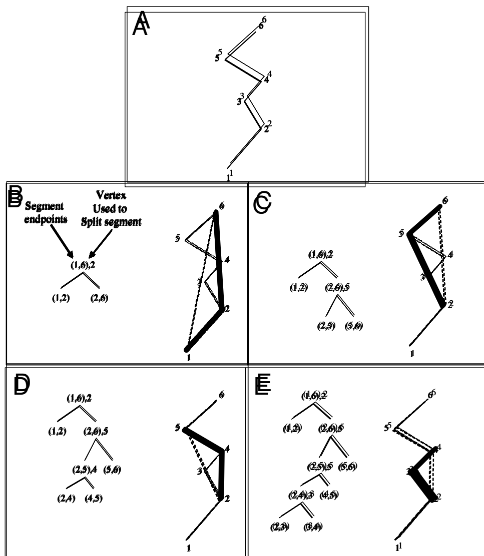

Figure 5: A-E illustrate the partial construction of a BLG-tree. The trees in B-E show all nodes added at successive tree levels. The polyline to the right of each tree illustrates the polyline geometry related to the newest tree node. Dashed lines are the spanning line segment being refined. Thick lines are the polyline refinement.

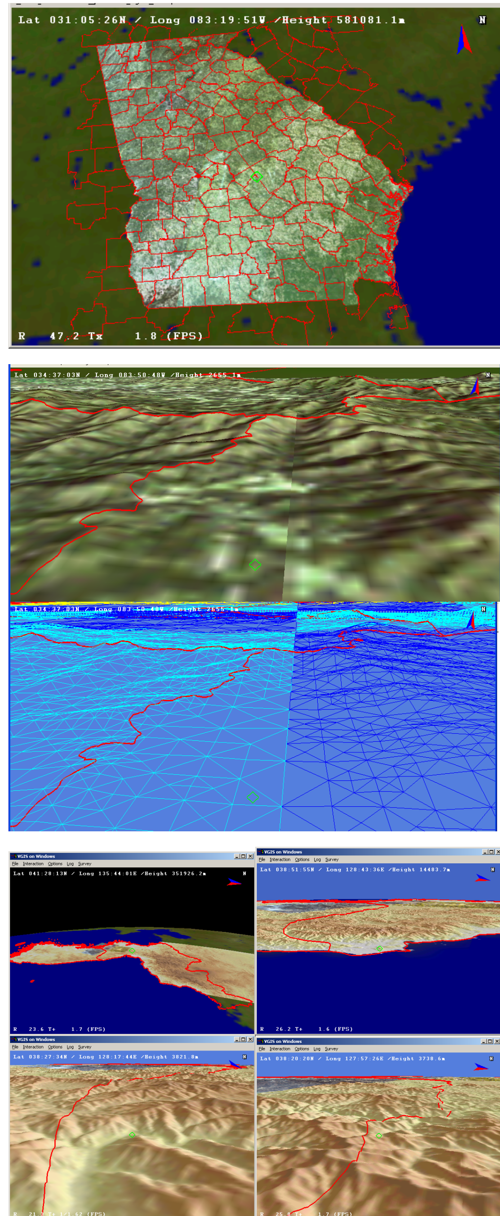

Figure 6: Georgia County Borders (240K points) and North & South Korea borders (10K points).

© The Eurographics Association 2003.