Terrain Representation using Right-Triangulated Irregular Networks

Abstract

A Right-triangulated Irregular Network, often called an RTIN, is a quadtree-based triangulation commonly used for hierarchical representation of terrain for 3D visualization. In this paper we provide a representation of RTINs, so that we are able to access neighbors in constant time. Further, we note that piecewise planar representations of surfaces are unable to fit curved surfaces within a triangle, which causes the triangles to split several times until the required accuracy is reached. We describe how RTINs can be used to represent terrains as a mosaic of polynomial surface patches. Finally, we compare memory requirements for two representations of RTINs (planar and quadratic) and a typical general TIN representation of a terrain, for a range of elevation errors.

1 Introduction

Terrain data in Geographic Information Systems is often sampled on raster grids, but displayed as collections of triangles. The Triangulated Irregular Network (TIN), a piecewise-planar approximation of the surface with a mesh of triangles, is often used because the data structure can adapt to the terrain—placing points on breaklines and using large triangles in areas of low variation—and because graphics pipelines are optimized for triangles.

A Right-Triangulated Irregular Network is a special TIN using isosceles right-angled triangles, formed as a binary tree by splitting [5] along the perpendicular bisector on the hypotenuse. Both RTINs and general TINs are irregular in structure, that is, they allow non-uniform sampling. Some parts of the terrain are represented with fewer triangles than others in such irregular meshes. Though an RTIN typically has more triangles for a given error-bound, it has clear advantages over other TIN variants for data sampled on a grid. The most important is that it is hierarchically structured like a quadtree [14], so it can avoid storing edge and neighbor information and derive this information from the hierarchy.

In this paper, we show that when triangles are assigned binary ids in the natural way according to this hierarchy, the ids of the neighbors of a triangle can be computed with $O(1)$ arithmetic operations. We also show how terrains can be approximated by piecewise-quadratic surfaces and compare the memory requirements for such a representation with other piecewise-planar representations.

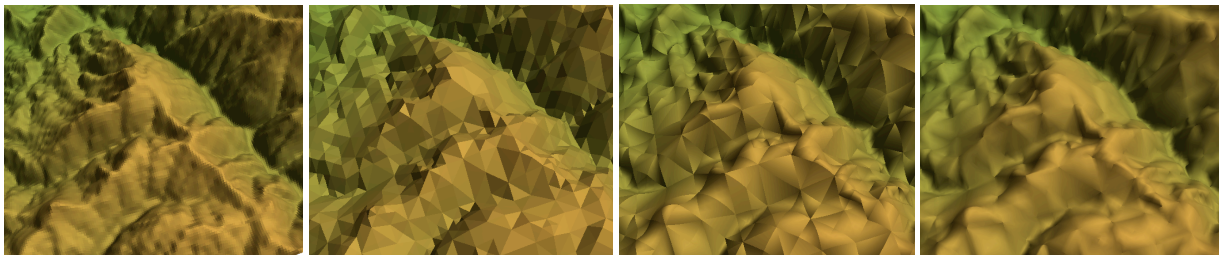
2 Related Work

There has been a lot of work done in approximation surfaces, in particular multi-resolution representation of spatial data as quadtree-like structures. RTINs have been independently developed in several large-scale terrain visualization systems [9, 4, 3, 10, 11]. Lee and Samet [7] discuss triangle quadtrees, which are quadtrees with equilateral triangles, each triangle splitting into four new triangles by joining the midpoints of the triangle edges. Equilateral triangles are encoded in a binary form so as to allow constant-time navigation to the neighbors. A similar approach is discussed by Lee et al in [8], to find equal-sized neighbors in a tetrahedral mesh. Samet [13] describes an algorithm to find neighboring terminal quadtree nodes by traversing up and down the tree, guided by the size and position of the nodes.

Lee with Samet and de Floriani [7, 8], as well as Evans et al [5] describe methods to compute neighbors of triangles in constant time. However, we provide a much simpler method than the above which draws on ideas from linear quadtrees initiated by Schrack [15]. Polynomials on triangulated terrain models have also been proposed before [12, 1, 2]. Preusser [12] describes the formulation of twice differentiable bivariate Hermite polynomials on a set of triangles. Akima [1, 2] describes the use of quintic bivariate polynomials to represent height fields over the xy plane.

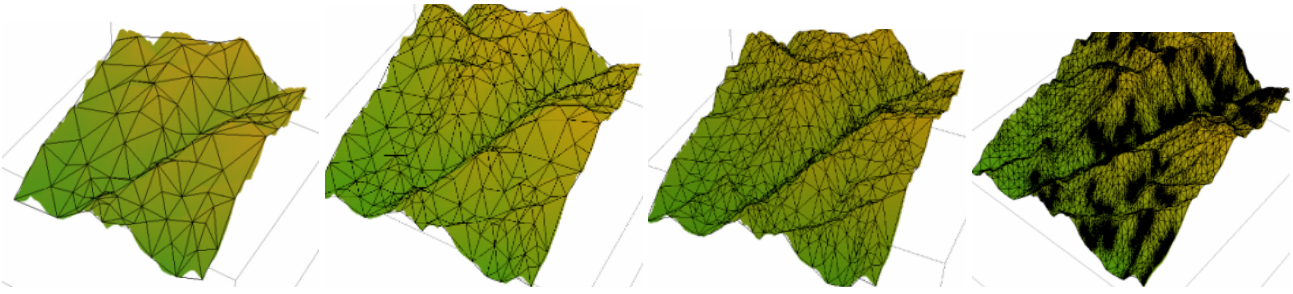
3 RTIN Representation

The key to an RTIN is to represent the hierarchical partition of the domain region of the xy plane. We store integer triangle `id`s in their binary representation. We begin with a large right triangle that contains the region of interest, the root ϕ. We split a right triangle T into two from its right angle to its hypotenuse, forming $T0$ to the left and $T1$ to the right. Figure 2 shows a labeled example.



(a) (b) (c) (d)

Figure 1. Various representations of the terrain: (a) input grid, (b) piecewise-planar representation, (c) piecewise-quadratic representation, (d) piecewise-quadratic representation with smoothed edge normals.



(a) (b) (c) (d)

Figure 3. Increasing levels of refinement in RTINs (on Lidar data over the China Lake region). The maximum average error per unit area within each triangle is set to (a) 20m, (b) 12m, (c) 8m, and (d) 3m, respectively.

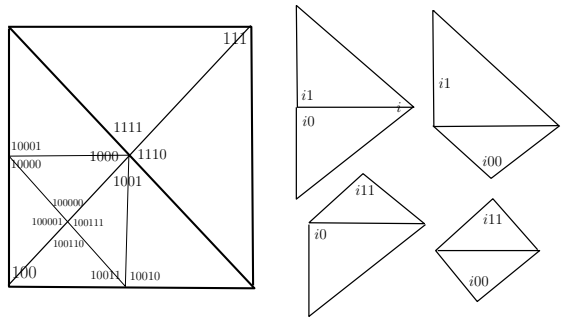


Figure 2. Portion of an RTIN and some neighbor cases. RTIN triangles' ids are at their right angles.

The above scheme ensures that the triangle `id` uniquely defines a path from the root to the triangle. We can also determine the location and orientation of the triangle from its `id`, if the coordinates for the root triangle is known. In fact, for every triangle we just store: (1) its `id`, (2) elevation of its north-west vertex (see Figure 4), and (3) coefficients (when using a polynomial representation). Storing one vertex per triangle helps reduce redundancy, and at the same time, we are able to cover almost all vertices in the mesh by choosing a suitable vertex for each triangle (see Figure 4). The vertices on the boundaries are likely to be left out (in our case, the south and the east boundaries) – so we store all 3 vertices of these boundary triangles.

Since every triangle `id` uniquely defines a path from the root triangle to itself, we do not need to store a tree structure. We represent the mesh as a list of triangles, and a hashtable with references to these triangles. A suitable function is chosen to hash triangle `id`s to the table. This allows constant time lookups for any given triangle (or its neighbor, as we will see below).

A triangle in a TIN can have neighbors sharing any of its three sides. Although we do not store edge pointers to neighbors in our mesh representation, it is desirable that we are able to find neighbors in constant time. We use a scheme which is motivated by the quadtree-like structure of the triangulation, where either the parents of two neighbors are neighbors themselves or the two neighbors have a common

parent. The following method describes how neighbors are calculated in $O(1)$ time.

As mentioned before, triangle `ids` are stored as binary strings. We use a postfix notation for the neighbors, with the letters $\{\mathbf{L}, \mathbf{R}, \mathbf{H}\}$, depending on whether it is sharing the left or right arm or the hypotenuse. Neighbors are at the same level of the hierarchy or one level higher (for $\mathbf{H}$) or lower (for $\mathbf{L}$ or $\mathbf{R}$). Actually, we need concern ourselves only with the same-size $\mathbf{H}$-neighbor: to find the lower neighbors for $\mathbf{L}$ (or $\mathbf{R}$), we take the left (or right) child and find its $\mathbf{H}$-neighbor; to find a larger neighbor (higher in the hierarchy), we just drop the least significant bit from the same-sized neighbor.

4 Computing an H-neighbor

In a structure of depth h, finding neighbors of triangles by recursion can take $\Theta(h)$ time, in the worst case. We claim the following simple algorithm computes the same-level $\mathbf{H}$-neighbor in $O(1)$ operations of AND $(\cdot)$, exclusive-OR $(\oplus)$, multiplication by 2, and addition (+).

HNEIGHBOR(T)

1. Assume $M = (10)^*$ is precomputed.
2. $D = (T \oplus 2T) \cdot M$.
3. $R = D + 2D$.
4. $C = R \oplus (R + 11)$.
5. Return $T \oplus C$.

We show a step-by-step execution of the pseudocode for an example HNEIGHBOR computation below.

HNEIGHBOR(01110010100101)
D =10000010101010
R =10000111111110
C =00001111111111
Return 01111101011010

Theorem: Given a triangle id T, HNEIGHBOR(T) correctly computes the id of same level $\mathbf{H}$-neighbor.

Proof: We derive this algorithm from the relations that Evans et al. [5] used to perform recursive neighbor calculations. In our notation, recall that a triangle T is split into children $T0$ on the left and and $T1$ on the right, so that $T0\mathbf{L} = T1$ and $T1\mathbf{R} = T0$. We also have recursive rules when a triangle and its neighbor are not direct children of their parent.

$$\begin{array}{ll} T0\mathbf{R} = T\mathbf{H}1 & T1\mathbf{L} = T\mathbf{H}0 \\ T0\mathbf{H} = T\mathbf{L}1 & T1\mathbf{H} = T\mathbf{R}0 \end{array} \quad (1)$$

These recursive rules apply until one meets the common ancestor triangle. If the root is reached first, we may have gone off the edge: $\rho\mathbf{L} = \rho\mathbf{R} = \rho\mathbf{H} = \emptyset$. By simply applying these rules, one obtains the neighbor in a tree of depth h using $\Theta(h)$ steps.

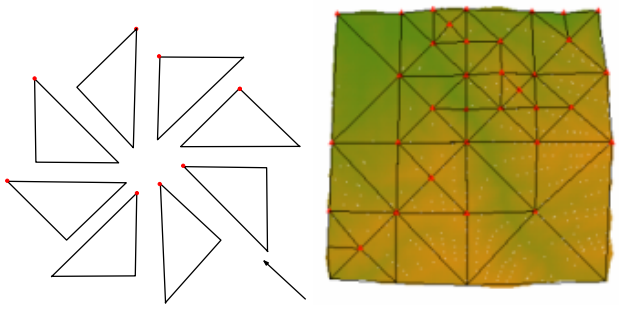
Notice that we can eliminate the $\mathbf{L}$ and $\mathbf{R}$ rules from 1 if we look two steps ahead:

$$\begin{array}{ll} T00\mathbf{H} = T11 & T11\mathbf{H} = T00 \\ T01\mathbf{H} = T\mathbf{H}10 & T10\mathbf{H} = T\mathbf{H}01 \end{array} \quad (2)$$

Thus, if we pair bits starting from the least significant one, the $\mathbf{H}$ carries through the first pair with 00 or 11, then stops. All bits carried through are complemented. Thus, the task is to compute the XOR mask $C = (11)^k$ for the appropriate $k > 0$. This is done in the following three steps.

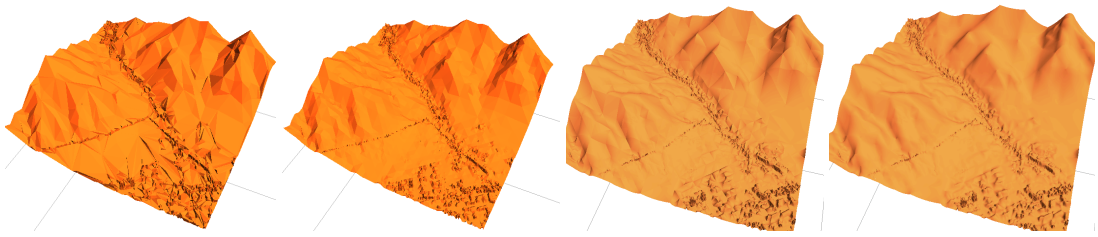
The computation of D marks which pairs contain two different bits by the pattern 10. Next, R fills in so a solid block of 1s indicate where the carry can ripple, extending one bit past, and containing a few spurious bits. Then C actually ripples a carry bit, and patches up both ends to obtain exactly $(11)^k$.

Since we can construct neighboring triangle ids, we can store triangles by simply hashing their ids. We recover the actual neighbors with at most two hash lookups, one for the neighbor at the same level and one for the parent. This can be a significant storage advantage over general TINs, which must record adjacencies.



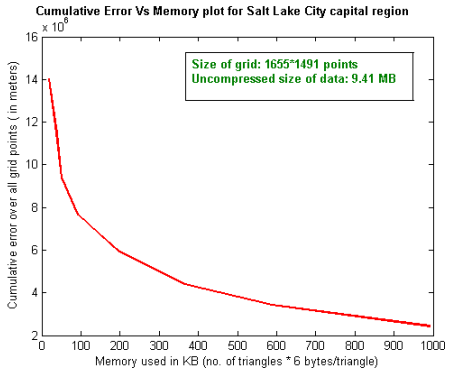
(a) (b)

Figure 4. (a) Highlighted points show the vertices whose elevations are stored. We store elevations of only one vertex (the NW vertex) per triangle. (b) Coverage of vertices in the mesh. Note that the boundary vertices (in the above figure, south and east boundaries), may not get covered – so we store all 3 vertices of these boundary triangles.

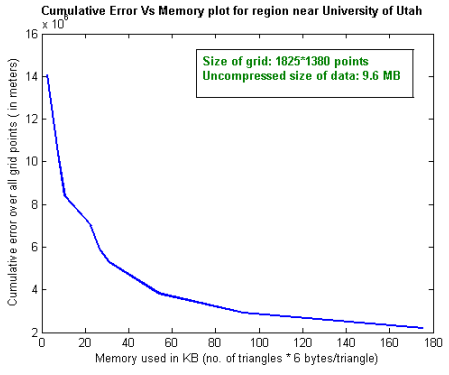


(a) (b) (c) (d)

Figure 5. Surface representation using regular TINs, and linear and quadratic polynomials over RTINs. Figure (a) shows a piecewise planar representation on regular TINs (9909 triangles), (b) shows the planar representation on RTINs (40,759 triangles), (c) and (d) show the surface as represented by piece-wise quadratic polynomials on RTINs (10794 triangles). Figure (d) has normals smoothed along triangle edges.



(a)



(b)

Figure 6. Plots of memory usage vs. Error: Planar Representation

5 Quadratic surface patches on RTINs

Typically, in a triangulated terrain, each triangle approximates a region of the surface covering *multiple* grid points. The number of grid points covered by the triangle would depend on the variation of the height field over the region where the triangle lies. Planar triangles are helpful in reducing storage (see plots in Figure 6), due to very low per-triangle storage cost. However, for a given error bound, a planar- representation ends up having lot more triangles than necessary, due to the inability to represent curved surfaces within triangles. We present a method, where *quadratic surface patches* are used to represent the terrain instead of planar triangles. In such a representation there is an overhead of storing coefficients with the triangles, but with benefit of reducing the overall number of triangles in the mesh.

The quadratic patch on a triangle would be represented by:

$$z = f(x, y) = c_0x^2 + c_1y^2 + c_2xy + c_3x + c_4y + c_5 \quad (3)$$

The coefficients vector $\mathbf{c} = \{c_i\}$, where, $i = 0, 1, \ldots, 5$, can be determined by a number of ways. We use two methods to calculate the coefficients – (1) by fixing vertices and (2) using a best-fitting polynomial. For (1), we determine the 6 coefficients by taking samples at the 3 vertices and 3 edge-midpoints. For (2), we take N sample points that lie within the triangle, and build up a $N \times 6$ matrix, $\mathbf{A}$, with corresponding values of $x^2, y^2, xy, \ldots$, and an elevation vector, $\mathbf{b}$, of size N. We then minimize the value of $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|$, to get the coefficients, $\mathbf{c} = \mathbf{x}_{min}$. Fixing vertices and edge-midpoints does not attempt to fit any of the interior points to the polynomial surface, but is a simple way to ensure that curves match along triangle edges. However, a best-fitting polynomial should fit the interior points to the surface more

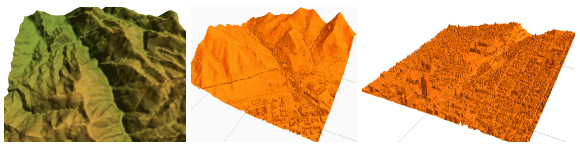
tightly, with the overhead of having to minimize a linear system with 6 variables and N constraints.

Once the coefficients are computed, we still need to render the piecewise polynomial surfaces. We subdivide each of the mesh triangles into smaller triangles, and use this finer mesh to render the polynomial surfaces. To avoid splits along edges due to difference in sampling, the edges of the smaller (rendered) triangles are made proportional to the edges of the original mesh triangles. Unit normals, $U(x, y)$, are computed on the surface $S(x, y, f(x, y))$ (from Equation 5) as follows.

$$\begin{aligned} U &= \frac{S_x \times S_y}{|S_x \times S_y|} \\ &= \frac{(-f_x, -f_y, 1)}{\sqrt{1 + f_x^2 + f_y^2}} \\ &\text{where, } f_x = \frac{\delta f}{\delta x} \text{ and } f_y = \frac{\delta f}{\delta y} \end{aligned} \tag{4}$$

To assign a single vector to points along triangle edges, the normals from all triangles sharing an edge are averaged, and the result is renormalized and assigned. This smooths the normals without changing the mesh representation in any way – it just gives a better visualization of the terrain represented by the piece-wise polynomial surfaces. Thus in Figure 5, (c) and (d) have identical error measures and memory requirements.

6 Experiments and Results



(a) (b) (c)

Figure 7. Input Grids: (a) China Lake region, (b) University of Utah region and (c) Salt Lake City capital.

Our goal is to compare memory usage vs. accuracy for several different representations. We have used both piecewise- planar and piecewise-quadratic representations on RTINs as described in the above sections and the planar representation on regular TINs for comparisons. The input grids used for our experiments were downloaded from the USGS Seamless Data Distribution System (http://seamless.usgs.gov). USGS provides data from both NED (National Elevation Dataset) and SRTM (Shuttle Radar Topography Mission) at various resolutions. We have used the NED $\frac{1}{3}$ arc second data for the region around China Lake and LIDAR (Light Detection and Ranging) data for the regions around University of Utah and Salt Lake City capital. We have cropped some of the data and converted them to raster DEMs (grids of squares) using the ArcInfo software.

We use a relatively small dataset from the China Lake region (Figure 7 (b)), and a larger grid from the region around University of Utah Figure 7 (b) near Salt Lake City. The metadata from the DEM files generated by ArcInfo are:

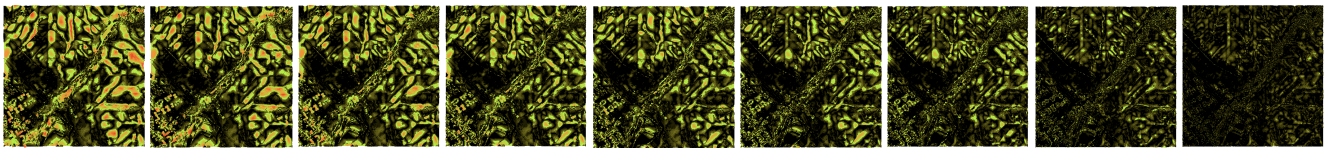
1. **China Lake Region:**
   ```
   ncols 401
   nrows 249
   xllcorner 373400
   yllcorner 3808200
   cellsize 9.2235465283486
   ```
2. **University of Utah Region:**
   ```
   ncols 1825
   nrows 1380
   xllcorner 429759
   yllcorner 4512752
   cellsize 1
   ```

The values above are in the UTM (Universal Transverse Mercator) coordinate system and distances are measured in meters.

We chose not to use SRTM data because of its low resolution. LIDAR data is higher in resolution, but at the same time is sensitive to aerosol and cloud particles, hence recording noisy data at times. The datasets over urban regions (for example, the region over Salt Lake City, Figure 7 (c)) are much more noise-prone, and result in undesirable number of subdivisions. In such cases it is hard to analyze the two representations that we have described in this paper. We have not used any noise-filtering or artifact-removal methods for our experiments, instead, have used regions with a fair amount of natural terrain features – like mountains and valleys.

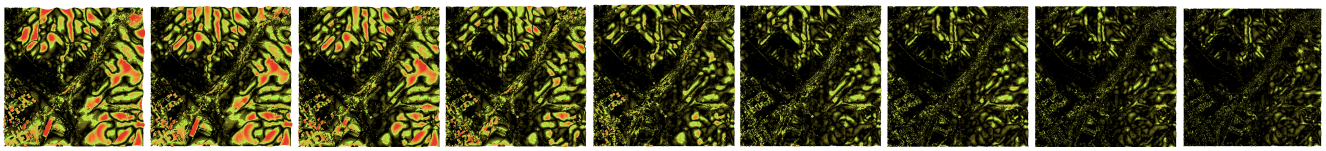
As described in the previous sections, each triangle contains its `id` and the elevation of one vertex in the piecewise-planar representation. The exact number of bytes per triangle is platform-dependent, for comparisons we have assumed 4-byte storage for both integers and floating points. Therefore a triangle (an average triangle, not those on the boundaries), can be stored in 8 bytes. For the piecewise- quadratic surface representation, we need 6 additional floating-point coefficients to be stored with the triangles. This increases the per-triangle size to $8 + 4 * 6 = 32$ bytes.

Figure 10 shows the triangle meshes obtained from the planar and polynomial representations. The polynomial representation has fewer number of triangles in the mesh



(a) (b) (c) (d) (e) (f) (g) (h) (i)

Figure 8. Error grids from the planar surface representation on RTINs. Grid from the University of Utah region is used as input. Each pixel of the grid is colored based on the *absolute error* at that point. As the absolute error increases, the color changes from black to green to yellow to red. Refer to Table 2 for corresponding memory usage for each of the above experiments.



(a) (b) (c) (d) (e) (f) (g) (h) (i)

Figure 9. Error grids from the quadratic surface representation on RTINs. Grid from the University of Utah region is used as input. Each pixel of the grid is colored based on the *absolute error* at that point. As the absolute error increases, the color changes from black to green to yellow to red. Refer to Table 3 for corresponding memory usage for each of the above experiments.

as expected. Considering the fact that each triangle in the polynomial representation needs 4 times more storage than the planar triangles, the reduction in the number of triangles still does not allow us to easily choose between the two representations. However, in some applications we might need to store a lot more information with the triangles (irrespective of the representation), for example, demographic data, vegetation, maximum or average slopes, drainage information, etc. It might then be reasonable to represent RTINs with polynomial patches with the additional storage of 24 bytes/triangle for coefficients, in order to reduce number of triangles in the mesh from 3837 to 1481 triangles (as in Figure 10).

For regular TINs, we have assumed the following memory structure: For a TIN with V vertices, the number of triangles are about $2V$ and the number of edges are around $3V$ (from Euler's equations). Each triangle would have 3 vertex pointers and 3 neighboring triangle pointers. So, assuming 4-byte storage for floating points, the memory required to store such a TIN would be about $(V * 3 * 4)$ bytes $+ (2V * 6 * 4)$ bytes $= 60V$ bytes. Allowing, an additional 4 bytes for flags, we need $64V$ bytes to store the TIN. We have used Scape (version 1.2), a terrain simplification software from CMU [6], to generate the TIN models from the raster grids.

The number of triangles in the RTINs and vertices in the regular TINs are chosen suitably so that the total memory utilization varies over a common range (about 25 kB to 750 kB). For each such representation, we record the sum of absolute errors over all grid points. Figures 8 and 9 show the variation of absolute error over the entire region.

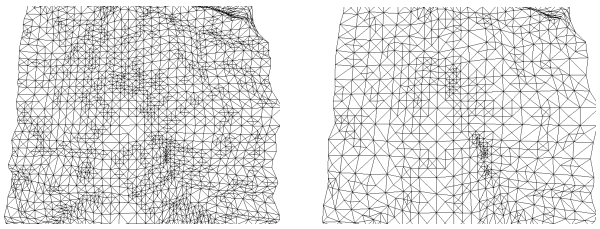


Figure 10. Triangle meshes for planar and quadratic representations respectively. There are 3837 triangles for the planar representation and 1481 triangles in the representation using quadratic polynomials.

Figure 11 shows plots of memory usage vs. cumulative error over the entire grid, for three representations of the terrain. Note that although the planar representation appears to be slightly better than the polynomial representation using RTINs, in applications where each triangle structure bulkier (with more terrain information) having fewer triangles (as in the polynomial representation) would be a more significant benefit.

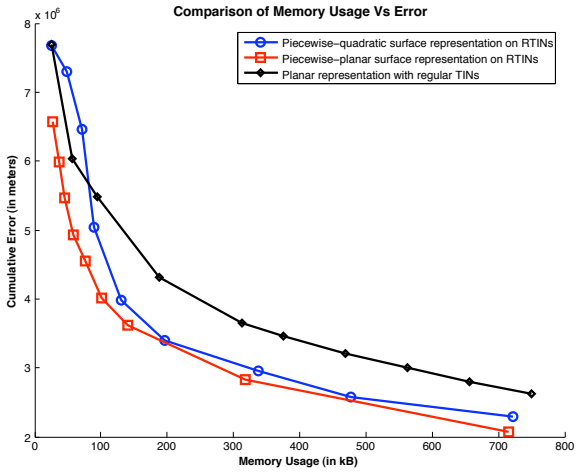


Figure 11. Comparison of memory usage vs. accuracy between various surface representations (values from Tables 1, 2 and 3).

SNo.	No. of Triangles	Memory Usage (in kB)	Total Error ($\times 10^6$ m)
(a)	820	25.781	7.69487
(b)	1765	55.430	6.02655
(c)	2963	92.883	5.47972
(d)	5933	185.930	4.31289
(e)	9909	310.367	3.64954
(f)	11896	372.563	3.46474
(g)	14869	465.680	3.20724
(h)	17863	559.289	2.99998
(i)	20848	652.688	2.79482
(j)	23844	746.344	2.62189

Table 1. Error Vs. Memory readings from experiments with the planar surface representation on regular TINs using University of Utah grid.)

7 Conclusions

In this paper we describe how to represent terrains as RTINs efficiently, and present a method for fast neighbor computations, which are useful in applications for traversals over the terrain. We also study the usefulness of polynomial surfaces with the underlying RTIN model, which allows us to represent the terrain using fewer triangles by fitting points to polynomial surfaces. Also, we have represented each triangle minimally with just enough geometric information to regenerate the terrain. Various applications might require triangles to store other information – like demography, drainage or vegetation – thus making each triangle bulkier to store. Reduction in the number of triangles becomes a significant benefit in such applications, and its reasonable to use polynomial patches with RTINs instead of planar RTINs.

RTINs provide a better representation than general TINs in hierarchical approximations such as multiple level-of-detail approximations. However, RTINs usually have larger number of mesh triangles than normal TINs for a given accuracy. This is due to the fact that the xy location of a point inserted in an RTIN is not data-dependent – it must lie on the midpoint of a hypotenuse to allow splitting of a right-angled isosceles triangle into two smaller ones. On the other hand, TIN has the advantage of allowing a point to be inserted at the "most important" location in the triangle.

8 Future Work

Using polynomial surface patches to represent terrains is also an area which needs further research. The main challenge is to find a way to fit interior grid points tightly to the polynomial surface. We have used only quadratic surfaces, but it might be useful to try cubic surfaces too. Using higher degree polynomials has the overhead of storing more coefficients, not necessarily guaranteeing a tighter fit to the grid points. In fact, polynomial patches of degree 5 or higher often tend to be noisy.

Rendering curved surface patches is another problem to consider. We have used a finer triangle mesh (not stored, but generated on the fly) just for rendering, but for large terrains, rendering speeds are quite slow. Although in this paper, our main goal was data representation and compression, it might be useful to try level-of-detail methods to render the curved surfaces to attain interactive frame rates.

References

[1] H. Akima. Bivariate interpolation and smooth surface fitting based on local procedures. *Commun. ACM*, 17(1):26–31, 1974.

[2] H. Akima. A method of bivariate interpolation and smooth surface fitting for irregularly distributed data points. *ACM Trans. Math. Softw.*, 4(2):148–159, 1978.

[3] L. de Floriani and E. Puppo. Hierarchical triangulation for multiresolution surface description. *ACM Trans. Graph.*, 14(4):363–411, 1995.

[4] M. A. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *IEEE Visualization*, pages 81–88, 1997.

SNo.	No. of Triangles	Memory Usage (in kB)	Total Error ($\times 10^6$ m)
(a)	3535	27.617	6.5667
(b)	4657	36.383	5.993
(c)	5688	44.438	5.4705
(d)	7384	57.688	4.9285
(e)	9766	76.1077	4.5543
(f)	12972	101.34	4.0127
(g)	17939	140.15	3.6255
(h)	40759	318.43	2.8245
(i)	91485	714.73	2.0768

Table 2. Error Vs. Memory readings from experiments with the planar surface representation on RTINs using University of Utah grid. (Refer to Figure 8(a)-(i).)

SNo.	No. of Triangles	Memory Usage (in kB)	Total Error ($\times 10^6$ m)
(a)	831	25.969	7.6716
(b)	1577	49.281	7.2971
(c)	2269	70.906	6.4545
(d)	2871	89.719	5.0385
(e)	4154	129.81	3.9807
(f)	6299	196.84	3.4035
(g)	10794	337.31	2.9537
(h)	15294	477.94	2.5837
(i)	23115	722.34	2.2860

Table 3. Error Vs. Memory readings from experiments with the quadratic surface representation on RTINs using the University of Utah grid. (Refer to Figure 9(a)-(i).)

[5] W. S. Evans, G. Kirkpatrick, and G. Townsend. Right-triangulated irregular networks. *Algorithmica, vol. 30, no. 2, pp. 264-286*, 2001.

[6] M. Garland and P. S. Heckbert. Fast polygonal approximation of terrains and height fields. *CMU Tech. Report*, Sep 1995.

[7] M. Lee and H. Samet. Navigating through triangle meshes implemented as linear quadtrees. *ACM Transactions on Graphics*, 19(2):79–121, 2000.

[8] M. Lee, H. Samet, and L. de Floriani. Constant-time neighbor finding in hierarchical tetrahedral meshes. In *SMI '01: Proceedings of the International Conference on Shape Modeling & Applications*, pages 286–295, Washington, DC, USA, 2001. IEEE Computer Society.

[9] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. *Computer Graphics (SIGGRAPH 96)*, pages 109–118, 1996.

[10] P. Lindstrom and V. Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transaction on Visualization and Computer Graphics*, 8(3):239–254, 2002.

[11] R. Pajarola, M. Antonijuan, and R. Lario. Quadtin: Quadtree based triangulated irregular networks. *Proceedings of IEEE Visualization 2002*, 00, 2002.

[12] A. Preusser. Algorithm 684: C1- and c2-interplation on triangles with quintic and nonic bivariate polynomials. *ACM Trans. Math. Softw.*, 16(3):253–257, 1990.

[13] H. Samet. Neighbor finding techniques for images represented by quadtrees. *Computer Graphics and Image Processing*, 18(1):37–57, Jan. 1982.

[14] H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[15] G. Schrack. Finding neighbors of equal size in linear quadtrees and octrees in constant time. *CVGIP: Image Underst.*, 55(3):221–230, 1992.