ROAM Using Surface Triangle Clusters (RUSTiC)

by

ALEX A. POMERANZ

B.S. (University of California at Davis) 1998

THESIS
Submitted in partial satisfaction of the requirements for the degree of
MASTERS OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES
of the
UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

Committee in Charge
2000

Abstract

Interactive visualization of terrain and general complex surfaces is a difficult problem that requires large data sets to be displayed and manipulated at high frame rates. Operations such as rotation and panning must be supported so a user can examine the data in critical areas, while maintaining highly accurate images. While previous surface approximation algorithms have yielded accurate and attractive-looking images, these algorithms are using only a fraction of the power of current graphics hardware due to being CPU bound. RUSTiC is an attempt to take advantage of the simplicity and accuracy of ROAM (Real-time Optimally Adapting Meshes), a previously developed surface approximation algorithm, while increasing the displayable polygon count by at least one order of magnitude.

Acknowledgments

This work was partially supported through an ASCI Level-2 contract awarded to the Center for Image Processing and Integrated Computing (CIPIC) at the University Of California at Davis. I would like to thank Mark Duchaineau at Lawrence Livermore National Laboratory for all of his great ideas, time spent fielding questions of mine, and time spent drawing ASCII art e-mails. Also, I would like to express my thanks to Ken Joy for drawing me into the field of scientific visualization, making it interesting, and giving me the chance to explore it at my own pace. Finally, I would like to thank all of my friends who supported me in so many ways through the entire process of being a graduate student.

Contents

Chapter 1

Introduction to RUSTiC

The collection and analysis of numerical data is an integral part of many scientific disciplines. For example, chemists collect and generate data on the structure of molecules, paleomagnetists collect data on geological and magnetic features of rocks, and physicists collect data on how gases interact under pressure. Unfortunately, interpreting numerical data is a difficult and time-consuming task for humans, especially when the data sets are large. To further complicate matters, as technology has improved, the complexity of the data to be analyzed has increased dramatically, leading to increasing difficulty exploring that data.

Although computers are not very good at analyzing data automatically, they are good at organizing and manipulating it. Visualization is centered around the idea that computers can be used to help organize complex numerical data into pictures that are more easily interpreted by humans. Various methods have been developed over the past twenty years to help visualize data with computers. For example, satellite technology allows us to scan the surface of the Earth and record the elevation of a geographic region (“height field data”). This data can be represented numerically as a two-dimensional (2D) array of elevations. A computer can be used to construct an accurate model of the landscape the data was taken from, allowing easy analysis of terrain features.

In most types of visualization, terrain visualization included, it is useful for a user to be able to examine the representation of data from any arbitrary viewpoint. Ideally, a user should be able to rotate and zoom into the data so he/she can locate and examine interesting points quickly, and there should be minimal delay between the computer updating the viewpoint. Typically, 30 updates (frames) per second are considered sufficient.

Computers are constrained by their CPUs and graphics hardware, used to update frames. Requiring 30 frames per second, a computer has 33 milliseconds to do whatever processing the algorithm requires and output a finite number of shapes before it has to start producing the next frame. For large data sets, there is not enough CPU or graphics hardware time to display the entire data set at full resolution between frames. One choice is to show fewer frames per second, but users are very sensitive to the responsiveness of a program and will become quickly frustrated if the program is not responsive enough. Also, reducing frame rates is an unacceptable choice in many situations such as military simulations, where the responsiveness and real-time updating is critical. The other choice is to reduce the geometric complexity of the data set so that the computer does not have to spend so much CPU or graphics hardware time to display the data. Algorithms that reduce the geometric complexity of the data are called surface approximation algorithms, and RUSTiC belongs to this class of algorithms.

Previously developed state-of-the-art surface approximation algorithms, such as ROAM [**?**], can render 6000 triangles per frame at 30 frames per second on a single R10000 SGI Onyx processor with an Infinite Reality graphics board, yielding smooth and high-quality terrain renderings. However, current Intel-based 80x86 processor derivatives with high-end graphics cards, such as the NVidia GeForce, are capable of displaying upwards of 10 million textured triangles per second. Thus, the ROAM method's 180,000 triangles/second is only a meager 1.8% utilization of the graphics hardware on such platforms. This implies that the ROAM algorithm is running out of CPU time in between frames, not graphics hardware time. RUSTiC is an attempt to take advantage of the simplicity and accuracy of the ROAM algorithm while increasing the displayable polygon count by at least an order of magnitude. The goals of RUSTiC are to yield more accurate and higher-quality images, while allowing a user to manipulate larger and more geometrically complex data sets.

Chapter 2

The RUSTiC Method

The RUSTiC method is based on the same triangle bintree as ROAM [**?**]. Figure 2.1 shows the first six levels of the basic ROAM triangle bintree. The root triangle $T = (v_a, v_0, v_1)$ is defined to be a right-isosceles triangle at the coarsest level (l = 1). At the next level (l = 2), T is split into the *left child* $T_0 = (v_c, v_a, v_0)$ and the *right child* $T_1 = (v_c, v_1, v_a)$, where v_c is the midpoint of the edge (v_0, v_1). The rest of the bintree is defined by recursively repeating this splitting process for the children.

Because any triangle can be split into exactly two right-isosceles *children triangles, or subtriangles* (we then call the original triangle a *parent triangle*), we can organize the triangles as a binary tree hierarchy. Figure 2.2 shows the first four levels of the triangle bintree and the bintree hierarchy formed by the triangles. Starting with a root number of $N = 1$, the left child is always numbered $2N$, and the right child is always numbered $2N + 1$.

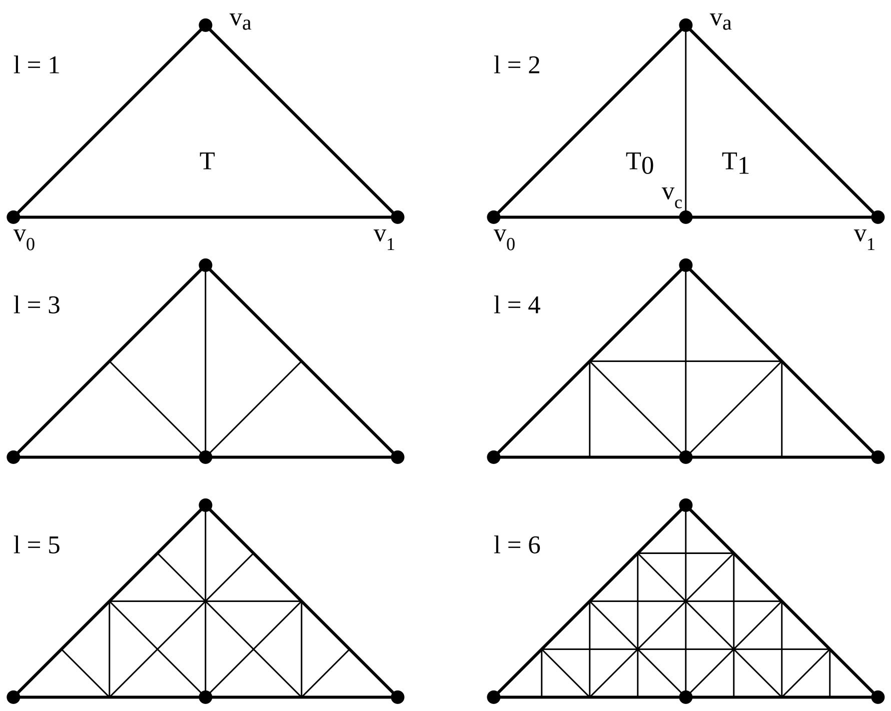

Figure 2.1: Levels 1–6 of triangle bintree.

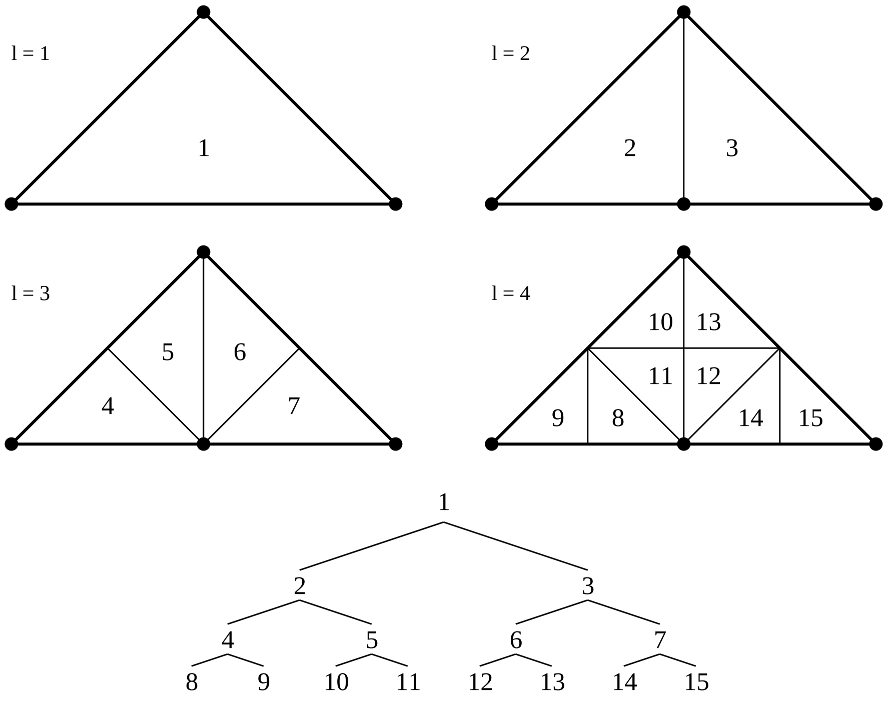

Figure 2.2: Levels 1–4 of triangle bintree and resulting bintree hierarchy.

A *ROAM triangulation* is formed in ROAM by assigning world-space coordinates to each bintree vertex. Given a 2D array of data points to be visualized, the array is "covered" by two root triangles, as shown in Figure 2.3. These root triangles serve as a coarse approximation of this particular region in the data.

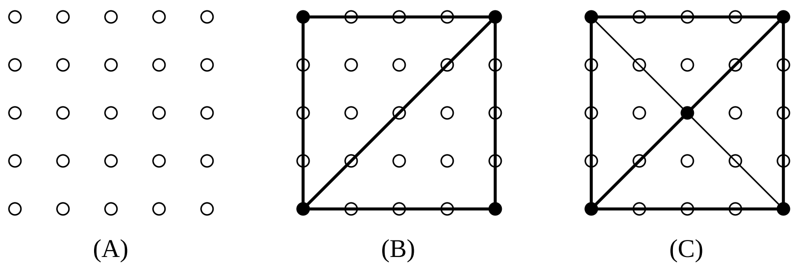

Figure 2.3: (A) shows a 5x5 grid of points. We note that these points do not need to be in a plane. (B) shows the region covered by two level 1 triangles. (C) shows the world-space covered by four level 2 triangles.

We note that, since the children triangles (e.g., T_0 and T_1) on level $l-1$ contain an additional vertex (v_c) amongst them over the parent triangle (e.g., T) on level l, the set of children triangles collectively yields a better approximation of a surface than the parent triangles. When we replace a parent triangle with its two children, we call this a split.

T_B is called the *base neighbor* of triangle T sharing the edge (v_0, v_1), T_L is called the *left neighbor* sharing the edge (v_a, v_0), and T_R is called the *right neighbor* sharing the edge (v_1, v_a). A simple split replaces triangle T with its children (T_0, T_1) and triangle T_B with its children (T_{B0}, T_{B1}). If the triangle has no base neighbor, only triangle T is split. This splitting method can be repeated recursively as long as a data point exists for v_c to obtain better approximations of some given surface. Merging of triangles works similarly. When triangle T and T_B belong to the same level, we call the pair (T, T_B) a *diamond*. A merge can be applied to a diamond (T, T_B) when all of the children of T and T_B (if T_B exists) are in the mesh. Figure 2.4 shows the basic split and merge operations.

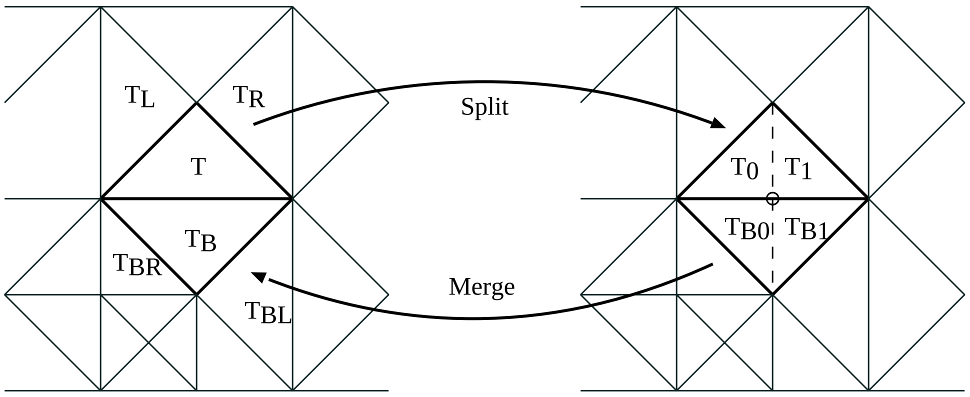

Figure 2.4: Split and merge operations on a bintree triangulation. Triangle T is split into subtriangles T_0 and T_1, and its base neighbor T_B is split into subtriangles T_{B0} and T_{B1}.

A triangle T on level l cannot be immediately split when its base neighbor T_B is from level $l + 1$. In this case, triangle T_B must be split first, which may require further recursive splits. This is called *forced splitting*, and Figure 2.5 shows an example of forced splitting.

This splitting and merging of triangles is the same as in ROAM, and RUSTiC makes use of the same split and merge queues to drive user-dependent optimization. ROAM starts from a set of root triangles, as shown in Figure 2.3, and uses a dual-queue system to decide which triangles to split to yield a better approximation, and which triangles need to be merged to decrease geometric complexity.

The difference between ROAM and RUSTiC lies in the construction and display of *triangle clusters*. ROAM starts with a set of root triangles, splits them into a set of continuous ROAM triangulation triangles, and then it outputs each triangle in that ROAM triangulation individually. RUSTiC follows the same process, except that a cluster of triangles is output instead of individual

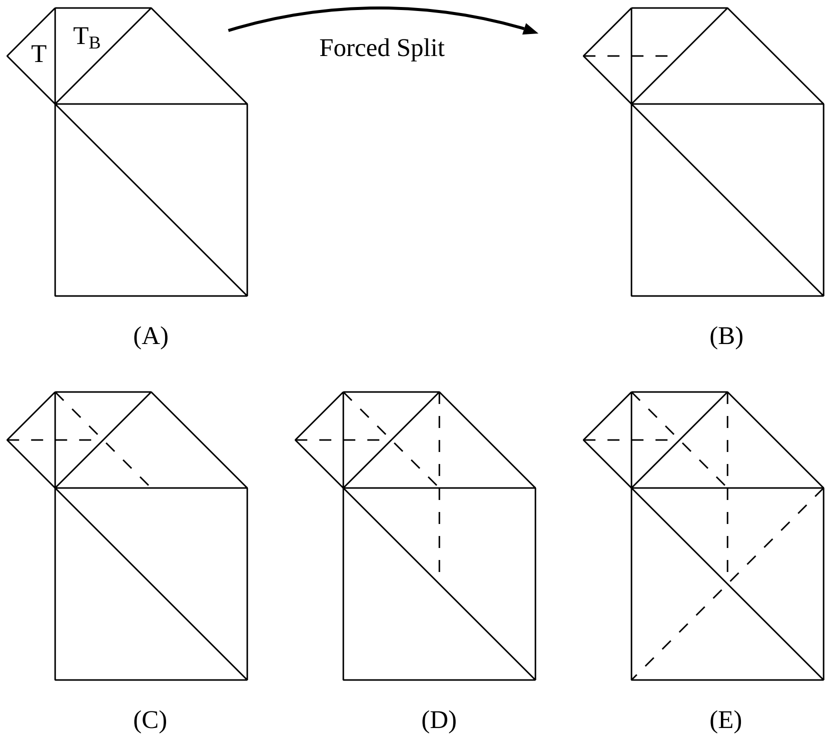

Figure 2.5: Forced split operation. (A) shows a region of adjacent triangles, including T and T_B. (B) shows T split. However, T_B must be split first, as shown in (C). But before T_B can be split, its base neighbor must be split, as shown in (D). The final state is shown in (E).

ROAM triangles, as shown in Figure 2.6.

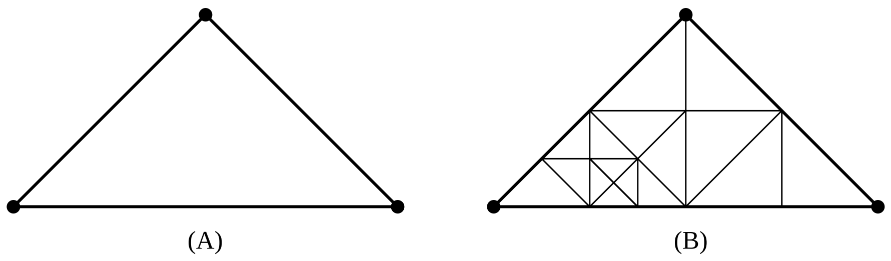

Figure 2.6: (A) Original ROAM triangulation triangle generated by ROAM. (B) is a pre-constructed cluster with 16 subtriangles that replaces (A) in the final mesh.

These triangle clusters must uphold the **edge constraint**: all possible adjacent clusters must have the same edge boundaries so that the resulting clustered surface (the *final mesh*) is always continuous. Figure 2.7 demonstrates the edge constraint by showing two adjacent mesh triangles and two adjacent clusters for those triangles. Care must be taken when dealing with the edge constraint, as any cluster edge can have two possible adjacent cluster edges. Figure 2.8 shows an example.

Any method may be used to construct the clusters as long as they adhere to the edge constraint. Although the clusters may be constructed and then displayed at runtime, this markedly slows down the algorithm. Since the bintree method of ROAM limits the location of vertices in the ROAM triangulation, a better method is to construct the triangle clusters for each possible triangle in the mesh in advance. This also allows the clusters to be “pre-packaged” in the most efficient method possible for “quick shipping” to the graphics hardware, as the speed of the cluster construction algorithm does not affect runtime speed. The number of triangles in each triangle cluster may be selected independently. Optimally, this number is defined so that the graphics hardware is always utilized.

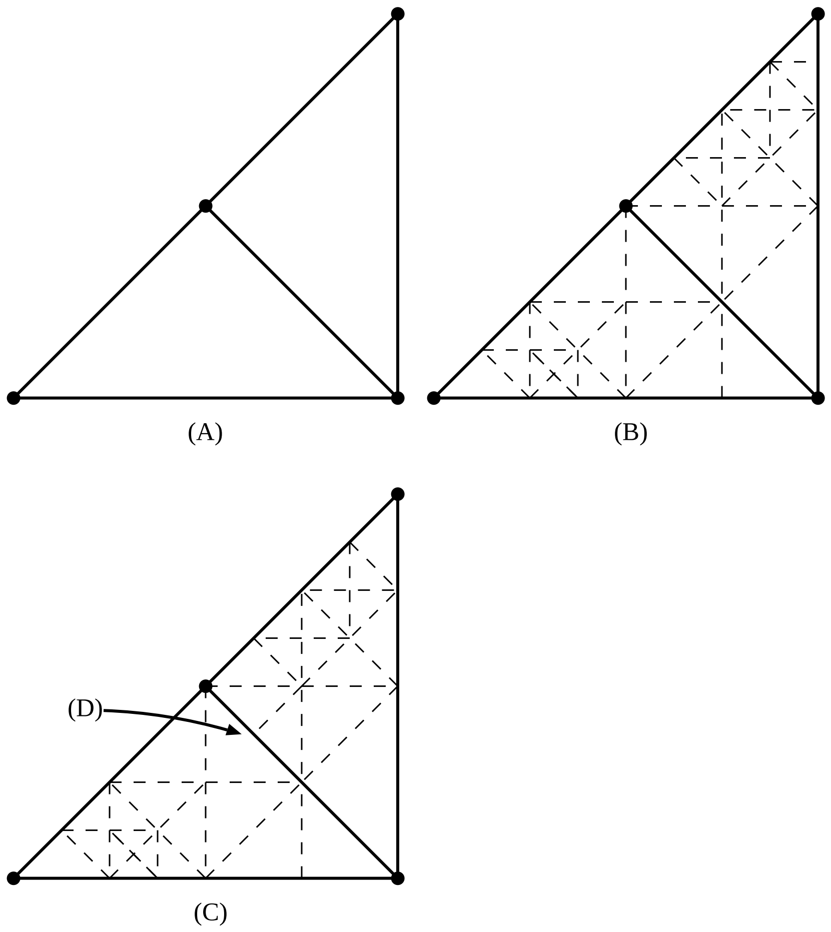

Figure 2.7: (A) shows two adjacent ROAM triangulation triangles. (B) shows the pre-constructed clusters for these triangles, each having 16 subtriangles. The adjacent edge is split in the same place by each. (C) shows two adjacent clusters that do not uphold the edge constraint along (D). The final mesh has a crack.

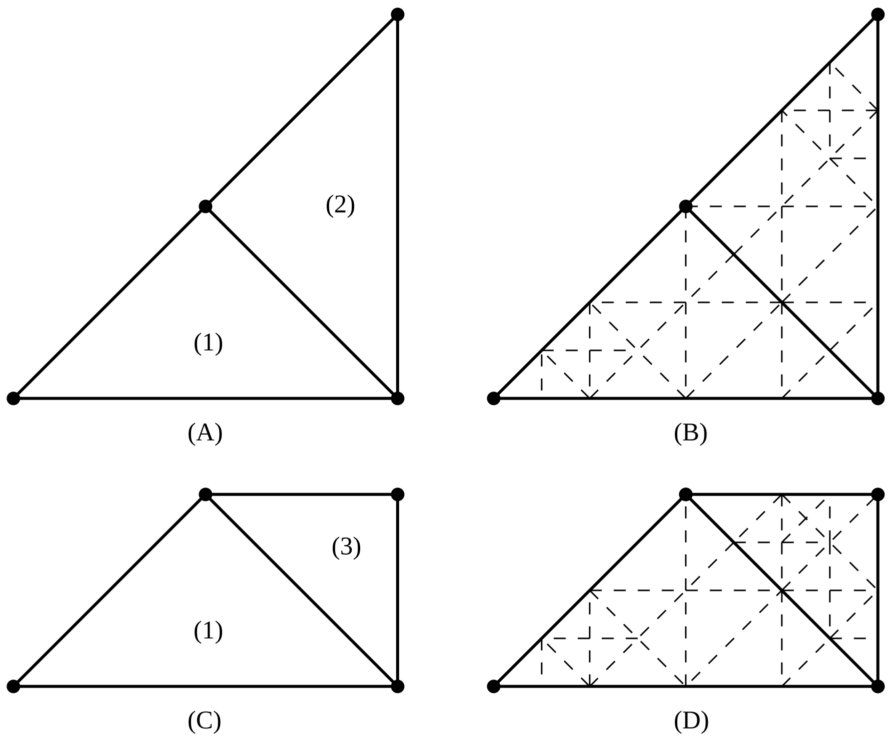

Figure 2.8: (A) shows triangle (2) adjacent to triangle (1) in the ROAM triangulation. (B) shows the pre-constructed clusters for triangles (1) and (2). (C) shows triangle (3) adjacent to triangle (1) in the ROAM triangulation. (D) shows the pre-constructed clusters for triangles (1) and (3). In both (B) and (D), the edge constraint is satisfied, despite the different pre-constructed clusters for (2) and (3).

Chapter 3

Constructing Clusters

We have devised three different methods for constructing clusters. Since ROAM-style split and merge operations are used in the construction of the ROAM triangulation, we leverage that framework in creation of the clusters. The non-trivial task during cluster construction is to always maintain the edge constraint.

The three methods are:

- Simple split, an easy to implement method of clustering.
- Adaptive split, a method that builds adaptive clusters quickly when you don't need to build clusters down to the finest level.
- Adaptive merge, a method that gives more precise control over cluster size and has a smaller memory footprint.

3.1 The Simple Split Method

One of the features of ROAM triangulations is that left and right neighbors of some triangle T are either from the same bintree level l as T or from the next finer level $l+1$. Base neighbors are either from level l or level $l-1$. Thus, all triangles in the ROAM triangulation are at most one level different than adjacent triangles, and the entire mesh is continuous. The most obvious and simple method of constructing clusters, the *simple split method*, is to split each ROAM triangulation triangle into N evenly sized subtriangles, where N = 2^Q and Q is an non-negative even integer.

When Q is an odd number, cracks in the mesh can occur: only the base edge of any triangle is split, and the base edge of one triangle might be the left or right neighbor edge of another triangle. This case is shown in Figure 3.1.

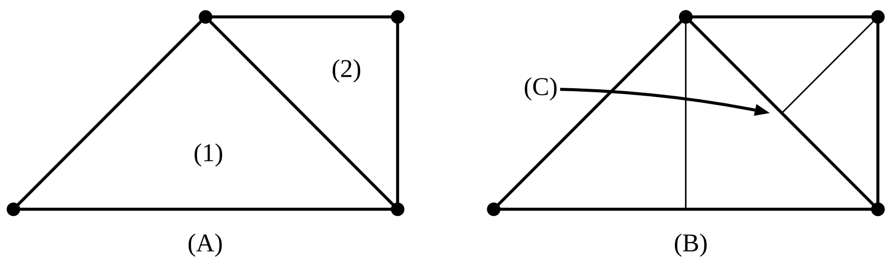

Figure 3.1: Example for Q = 1, N = 2. (A) shows two triangles in the ROAM triangulation. (B) shows the clusters that result. There exists a crack at (C).

When Q is even, each edge of the original base triangle is split exactly $2^{Q/2}-1$ times. When the individual triangles in the ROAM triangulation are replaced with these triangles, the surface will remain continuous because all edges have been split the same number of times. Figure 3.2 shows an example of this for $Q=2$.

Given a cluster size of 16 $(Q=4)$ and using the ROAM renderer, we can output 6000

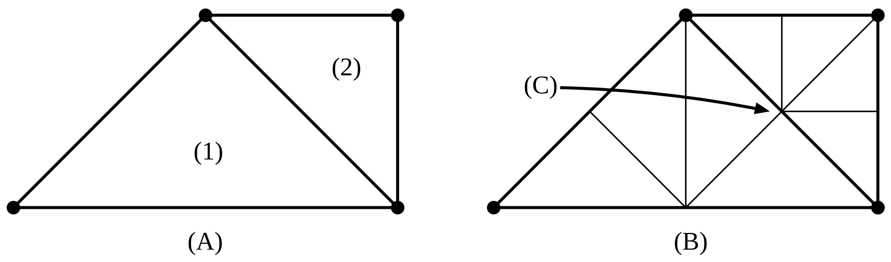

Figure 3.2: Example when Q = 2, N = 4. (A) shows two triangles in the ROAM triangulation. (B) shows the clusters that result. The edge at (C) is split at the same place on both sides.

base triangles times 16 triangles per cluster times 30 frames/second = 2.88 million triangles/second. This is a 16-fold increase in the number of triangles over the original ROAM algorithm.

However, it should be noted that this method of cluster building is non-adaptive, as it does not take into account which cluster triangles have the greatest errors and are prime candidates for splitting. A great deal of overall adaptivity still remains, since the ROAM triangulation that ROAM constructs is still done adaptively. Another downside of this method is the coarse control over the number of triangles per cluster, since the choice of N is limited to powers of two.

3.2 The Adaptive Split Method

The adaptive split method improves upon the simple split method by both adding adaptive behavior and increasing the level of control over the number of triangles per cluster. This yields clusters that not only look better, but are better approximations of the surface. Like the simple split method, we begin with an individual ROAM triangulation triangle. However, we are free to split this triangle as many times as we like and in any order we like, using the normal ROAM split method, so long as we adjust the clusters to ensure a continuous surface.

Any triangle on level l can have up to two adjacent clusters per cluster edge (one on level l and one on either level $l + 1$ or level $l - 1$ depending on the edge), as shown in Figure 2.8. When we split across one of these cluster edges, we need to ensure that both clusters are updated or a crack in the surface will result. The adjacent cluster on level l will be split automatically by the normal ROAM split algorithm. However, the other cluster must be split explicitly, and we do this by propagating the split up or down to that cluster level. An example is shown in Figure 3.3.

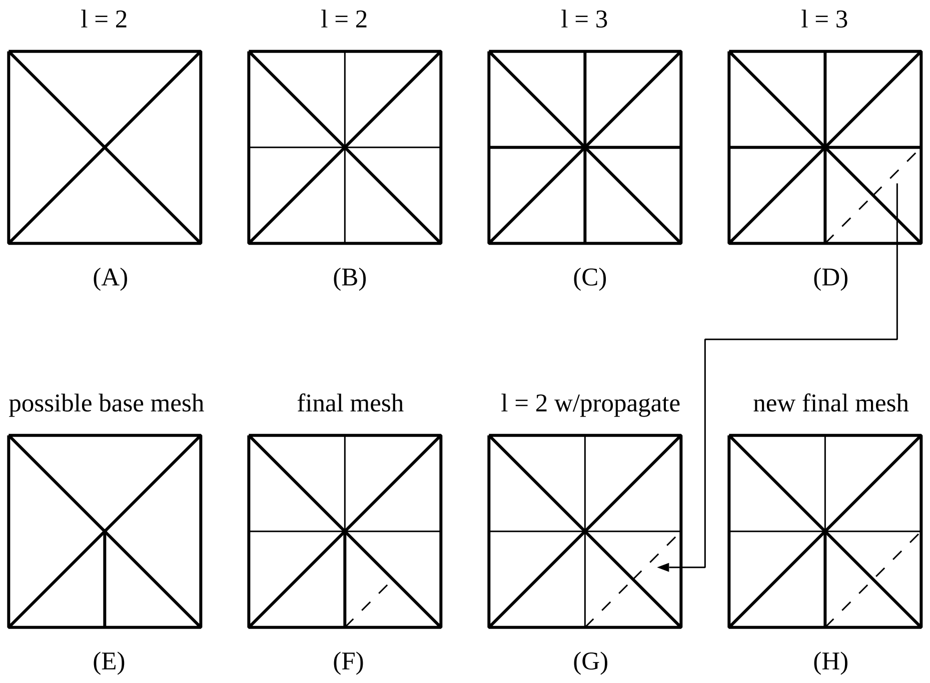

Figure 3.3: (A) shows four triangle clusters on level l = 2. (B) shows each of these clusters split once. (C) shows eight new triangle clusters on level l = 3. (D) shows one of these triangles split once (the other remain unsplit for simplicity). (E) shows a possible ROAM triangulation generated by ROAM. We note that triangles from levels l = 2 and l = 3 appear. (F) shows what happens when replacing the triangles in (E) with the corresponding triangles from (B) and (D). We note that there are two places where cracks will now appear. The solution is to propagate the split from level l = 3 (D) to level l = 2 (B). Level l = 2 now looks like (G) after the split has been propagated. (H) is the resulting final mesh from (E) with no cracks.

The easiest way to implement this method is to label the edges of all triangles according to the level of first triangle they appear in. In the initial triangles on level l = 1, the left edge label (E_L) and right edge label (E_R) are set to one, and the base edge label (E_B) is set to zero. Figure 3.4 shows the labeling scheme for a few levels.

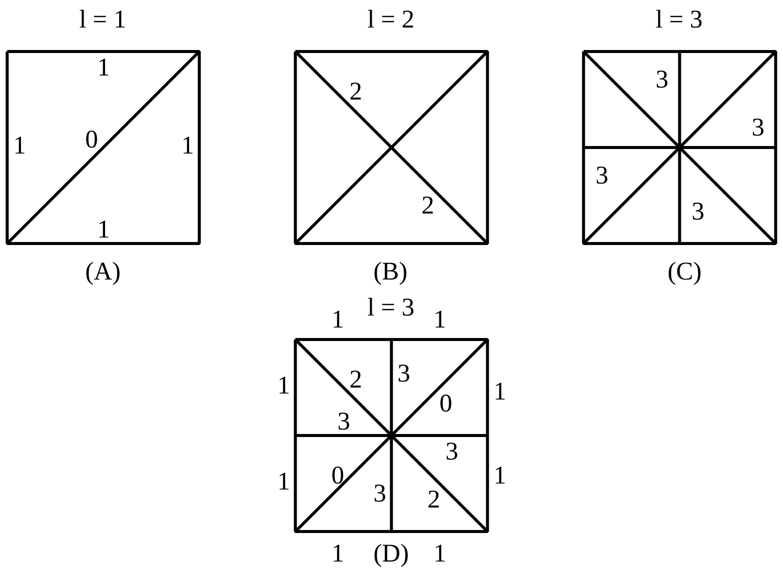

Figure 3.4: (A) shows the initial labels for two triangles at level l = 1. (B) shows the new labels for the edges added at level l = 2. (C) shows the new labels for the edges added at level l = 3. (D) shows the labels for all edges at level l = 3.

We note that, for any given triangle T on level l, the cluster edges will always have edge labels $<= l$, while interior edges (edges that are not on the cluster boundary) will have edge labels $> l$. Also, (E_L) and (E_R) have the same parity as l, while (E_B) has different parity from L. When a triangle is split across a cluster edge, the split needs to be propagated up to the next coarser level ($l - 1$) if the split is across the base edge, or down to the next finer level ($l + 1$) if it is across a side edge. This propagation scheme is summarized in Table 3.1.

For an example, consider the propagation shown in Figure 3.3: in step (D), the splitting of the triangle on cluster level l = 3 splits the edge with label e = 2 (from Figure 3.4). Thus, the cluster

cluster level $l >=$ edge level	parity(cluster level) $==$ parity(edge level)	propagate down to level $l + 1$
cluster level $l >=$ edge level	parity(cluster level) $!=$ parity(edge level)	propagate up to level $l - 1$
cluster level $l <$ edge level		no propagation

Table 3.1: Various propagations necessary to maintain crack-free clusters.

level $>=$ the edge level. Also, since 3 is odd and 2 is even, parity(cluster level) $!=$ parity(edge level). Looking at Table 3.1, it lists this case in the second line, and thus we know we need to propagate this split up a level (to level l = 2).

When creating all the clusters in advance, a simple procedure can be used. The steps of this procedure are:

1. Start on level l = 1, and split the clusters to the desired number of triangles N (making sure to split each triangle at least once). Any propagation up or down may be ignored since no other triangle clusters on other levels exist at this time.

2. Copy the structure of the triangles on level l = 1 to the structure of the triangles on level l = 2, as shown in Figure 3.5.

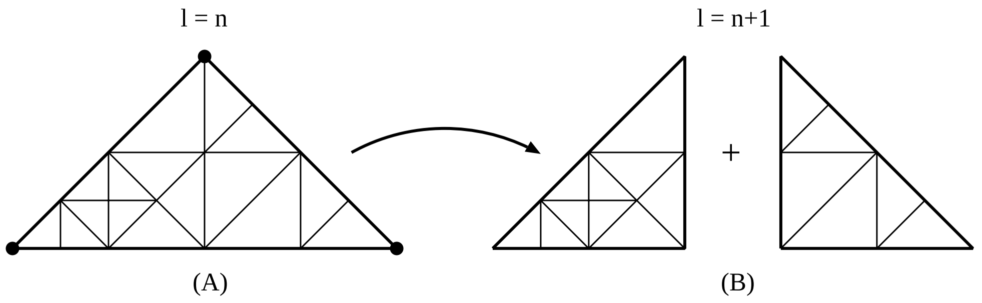

Figure 3.5: (A) shows a cluster for some triangle on level n (B) illustrates how that cluster is copied into two new triangles on level $n + 1$. The cluster in (A) is split down the center, resulting in the two subclusters shown in (B).

3. Split the triangles on level l = 2 to the desired triangle count N. We note that the triangles on

level l = 2 may already have some subtriangles resulting from step 2. Also, one has to handle upward propagation, since triangle clusters on level l = 1 now exist. However, downward propagation to level 3 may be ignored.

4. Copy the triangles on level l = 2 to the triangles on level l = 3 in the same manner as before.

5. Repeat steps 3 and 4 for each level until the finest level in the data set is reached.

At first glance, it may seem that “propagate down” does not need to be implemented. However, when creating the clusters on level $l = n$, one might propagate a split up to level $l = n - 1$, traverse across a couple of triangles on level $l = n - 1$ (via a ROAM forced split), and then propagate a split back down to level $l = n$. An example of this is shown in Figure 3.6.

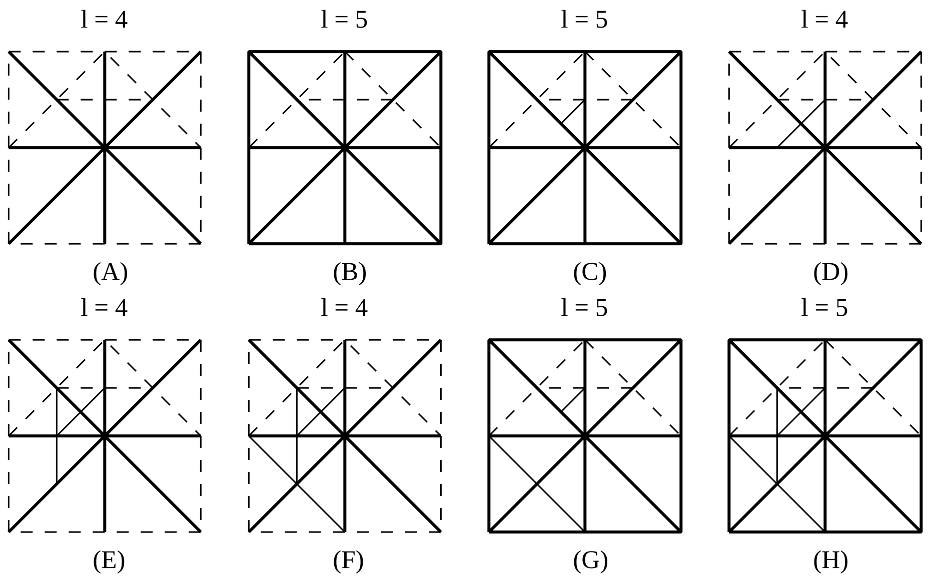

Figure 3.6: (A) shows some clusters built for a few half-triangles on level l = 4. (B) shows the structure copied to level l = 5. (C) shows a split on level l = 5. This split propagates up to level l = 4 as shown in (D). We have a forced split in (E) that propagates up to level l = 3. We have another forced split in (F) that propagates down to level l = 5 as shown in (G). Once all propagations are done, one can do the forced split on level l = 5, as shown in (H).

It is also possible that a split needs to be propagated, but the triangle to be split in the destination cluster has not been created yet. An example of this is shown in Figure 3.7. This can be easily solved: although the triangle has not been created yet, it is guaranteed that the triangle's parent exists. The parent can be split, yielding the child triangle that is the target of the propagated split.

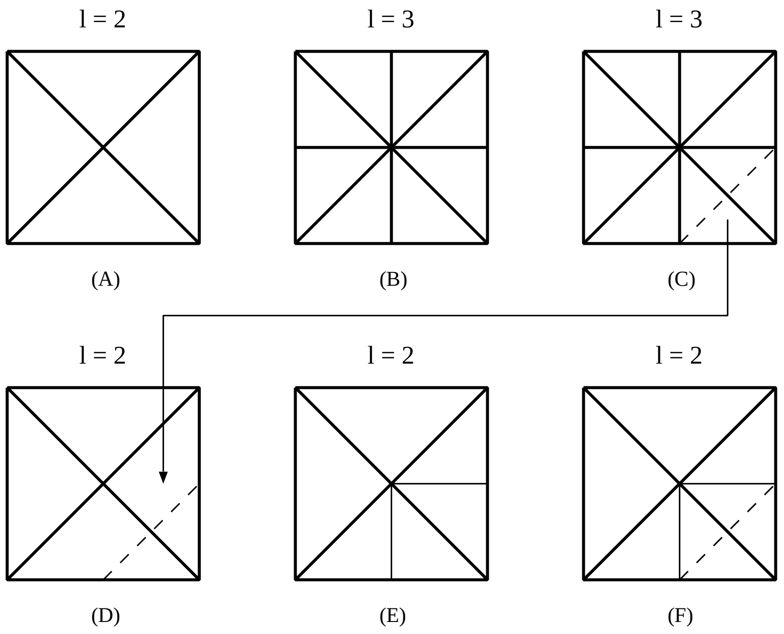

Figure 3.7: (A) shows some unsplit clusters on level l = 2. (B) shows some unsplit clusters on level l = 3. If we were going to split one of the clusters on level l = 3, such as in (C), it would propagate up to level l = 2 as shown in (D). But we cannot split the clusters on level l = 2 in this fashion. First, we have to split the clusters on level l = 2 as shown in (E), then we can propagate the split from (C) up to level l = 2 as shown in (F).

The adaptive split method has a few disadvantages to it. First, although each cluster has a target number of triangles N that it is split to when the cluster is being created, propagation may cause additional splitting in the cluster, resulting in more than N triangles per cluster. Experiments show that, when trying to achieve a target size of N triangles per cluster, where $N = 2^Q$ and Q

is an integer, the typical cluster size after all propagation is approximately $2N$ ($Q = 2, 4, 6, ...$), or approximately $4N$ ($Q = 1, 3, 5, ...$). This suggests adaptivity increases dramatically for $N = 32, 128, 512, ...$ triangles per cluster. Table 3.2 shows the average number of triangles per cluster for one particular experiment given different target values of N.

cluster on level	1	2	3	4	5	6	7	8	9
N = 8	15.5	12	9.88	8.78	8.29	8.1	8.02	8	8
N = 16	35.5	33.25	32.375	32.0625	32	32	30.02	15.01	8
N = 32	52.79	41.34	35.56	33.27	32.35	32.05	32	16	8
N = 64	132	130	128.88	128.19	121.12	60.56	32	16	8
N = 128	167.27	144.16	132.48	128.59	128	64	32	16	8

Table 3.2: Average number of triangles for all clusters on a given level, given a target of N triangles per cluster using the adaptive split method.

The second drawback to the adaptive split method is that all clusters must be stored in memory (or paged to disk) because splits can be propagated up or down at any time. This means the memory footprint of this method is quite large for large data sets.

The primary advantage of using the adaptive split method is that the computation of adaptive clusters is fast if you do not need to generate clusters down to the finest level.

3.3 The Adaptive Merge Method

Although the adaptive split method is better than the simple split method, the recursion makes it difficult to implement. The adaptive merge method yields the same benefits as the adaptive split method without all the recursion. Whereas the adaptive split method is based on splitting individual triangles, the adaptive merge method deals with merging diamonds. When two triangles T and its base neighbor T_B are on the same level l, the pair (T, T_B) is called a *diamond*.

Given a target number of triangles per diamond N, where $2^{Q-1} <= N <= 2^Q$ and Q is an integer, we can begin tiling our data set with *cluster diamonds* (or half-diamonds along a boundary) that contain exactly 2^Q evenly-sized interior triangles. At this state, the mesh is continuous, as all adjacent cluster diamond edges are split in the same places. Figure 3.8 shows an example.

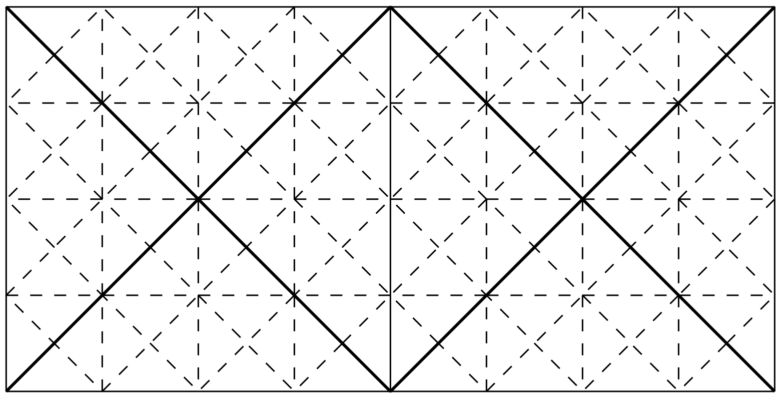

Figure 3.8: A diamond (in the center) and adjacent half-diamonds, split into to 32 evenly sized triangles.

We note that we may merge the interior diamonds of any cluster diamond, provided that the merge does not cross a cluster diamond edge and the mesh will still remain continuous. The same labeling scheme used in the adaptive split method can be used to determine which edges are cluster diamond edges. When E_B has the same parity as l and $E_B >= l$, that edge is a cluster diamond edge. Figure 3.9 shows an example of a triangle within a diamond that may be merged and one that may not be.

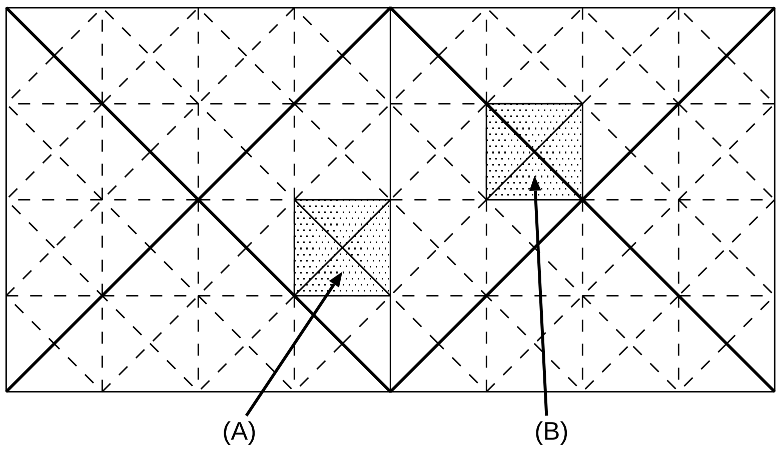

Figure 3.9: The cluster diamond in the center has cluster diamond edges (boldface). (A) is an internal diamond that may be merged since it does not cross the cluster edges. (B) is a diamond that may not be merged since it crosses the cluster edges.

Once we are satisfied with our merging the interior diamonds of a diamond cluster, we move up to the next coarser level of diamonds, copying the structure of the current level. Figure 3.10 shows this operation.

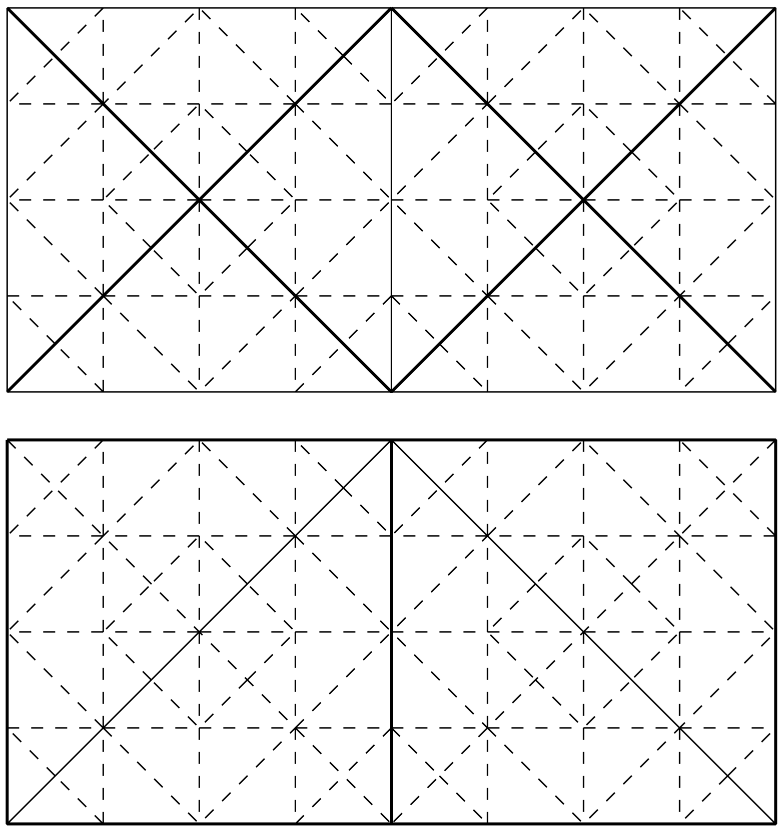

Figure 3.10: The top configuration is the mesh from Figure 3.9 with each diamond merged to 24 subtriangles. The bottom configuration is the result of copying the structure from one level of clusters to the next coarser level. We note that all the subtriangles stay the same, only the cluster boundaries move.

The edges that were unmergable in clusters on the previous level become mergable on this level since they are now internal base edges. Also these clusters will typically consist of more than N triangles. It is thus desirable to merge internal diamonds (not crossing the cluster diamond edges for the diamond clusters on this level) until the target of N triangles is reached. Figure 3.11 shows an example of this.

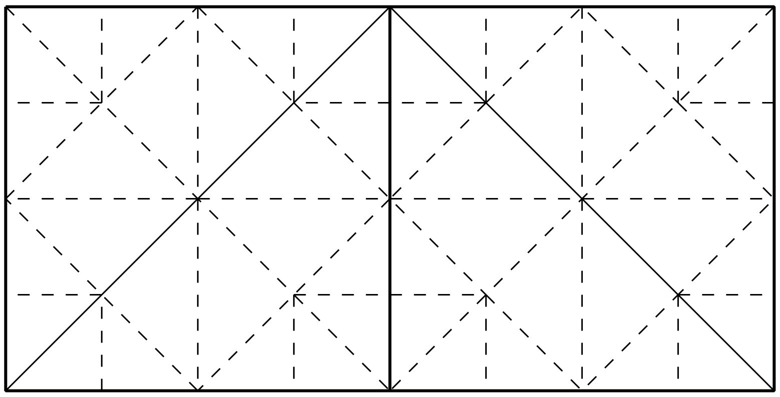

Figure 3.11: Mesh shown in Figure 3.10 (bottom) reduced to N = 24 subtriangles.

The sequence of steps for this method:

1. Tile the data set with diamonds such that each diamond has 2^Q triangles. Merge interior diamonds until one has N triangles, or until no interior merges are possible. No merging over the cluster diamond edges is allowed.

2. Create the diamonds on the next coarser level, copying the structure of the current level into the new diamonds.

3. Merge interior diamonds on the new level until one has N triangles.

4. Repeat steps 2 and 3 until the coarsest level.

The adaptive merge method has several additional advantages over the adaptive merge method. First, one is are always guaranteed approximately N triangles per cluster since there is no propagation introducing additional splits after the cluster has been created. Second, once a given level of clusters is created, it can be saved to disk and the memory freed. Thus, the memory footprint of this method is much smaller than the adaptive split method.

Chapter 4

Results

One useful result is a measurement how much RUSTiC improves the accuracy of ROAM. Table 4.1 shows some error results computed for one case using the Adaptive Split method. At this time, we have not implemented an optimized runtime view-dependent display environment. Results from doing so will be available when RUSTiC is published in a professional journal.

# of triangles/clusters	cluster size	# final triangles	ROAM error	RUSTiC error
100	16	4151	739.625	320
200	16	5563	460.75	276
300	16	8854	427.375	256
400	16	9879	347.25	201.5
500	16	10928	308.375	201.5
1000	16	14663	214.875	166
100	32	8602	739.625	256
200	32	11709	460.375	203
300	32	13706	427.375	189.5
400	32	15069	347.25	189.5
500	32	16549	308.375	173
1000	32	20719	214.875	125
2000	32	25077	145.875	83.5
3000	32	26282	115	61
4000	32	27530	92	61
100	64	19861	739.625	138.5
300	64	23744	427.375	138.5
500	64	25393	308.375	91.5
1000	64	28711	214.785	61
2000	64	30196	145.875	45
3000	64	30688	115	27
4000	64	31281	92	27
1000	128	29428	214.875	31.5
2000	128	30539	145.875	25
3000	128	30985	115	19

Table 4.1: Error statistics for ROAM vs. Rustic.

4.1 A 2.5D Example

Our algorithm was first applied to a United States Geographical Survey (USGS) data set of Roswell, New Mexico. Figures 4.1 through 4.4 show a sample ROAM-generated ROAM triangulation and the RUSTiC-clustered final mesh equivalent. The textures are colored based on the height and are not shaded.

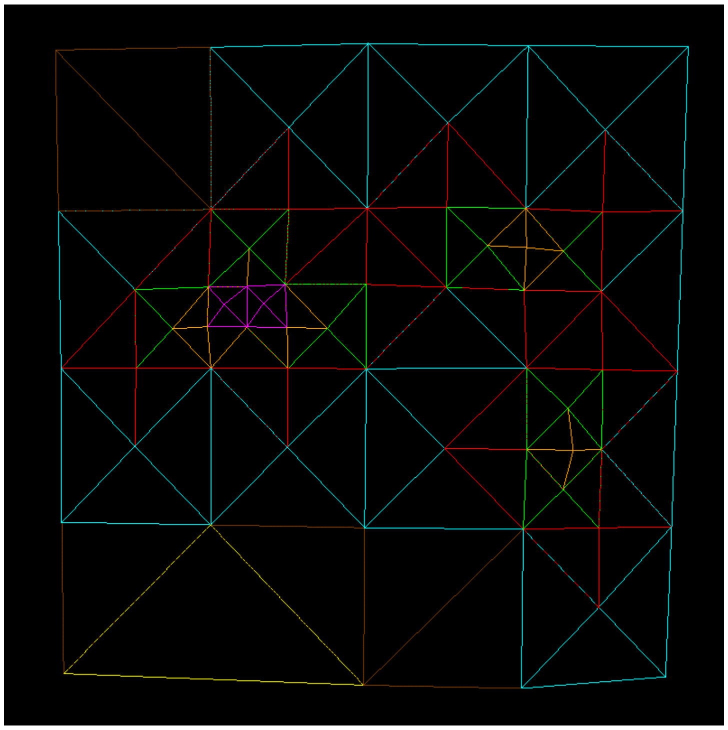

Figure 4.1: Base mesh constructed for Roswell data set.

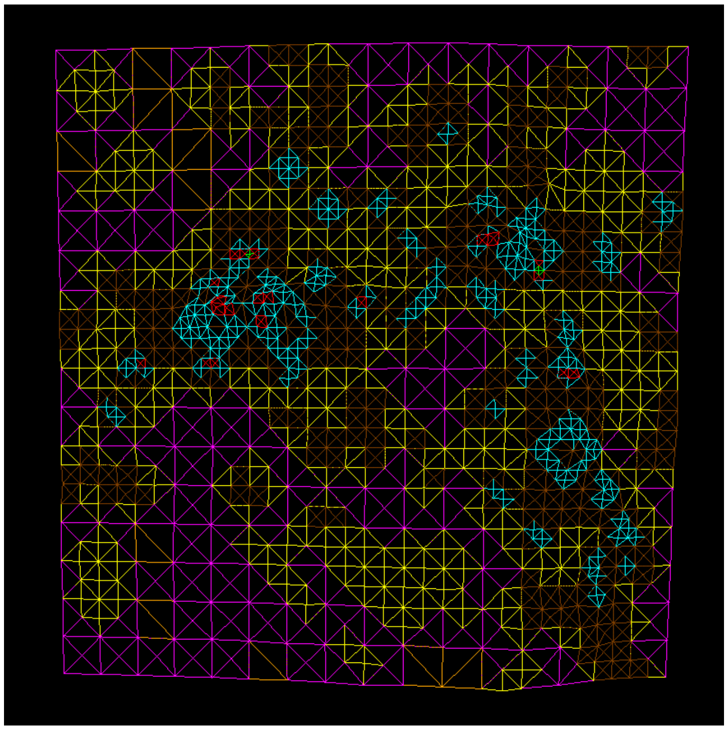

Figure 4.2: Final mesh constructed using clusters for Roswell data set.

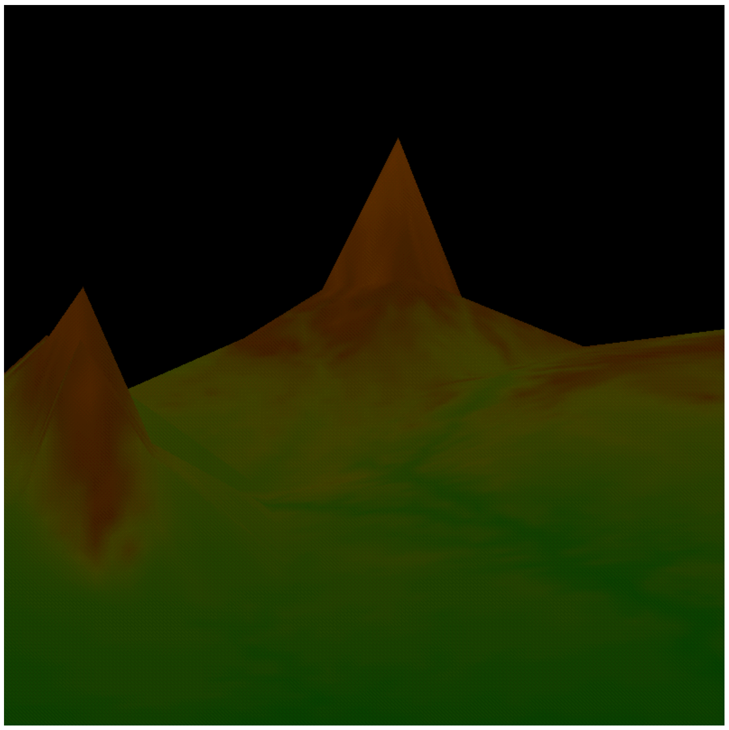

Figure 4.3: Base mesh with texture.

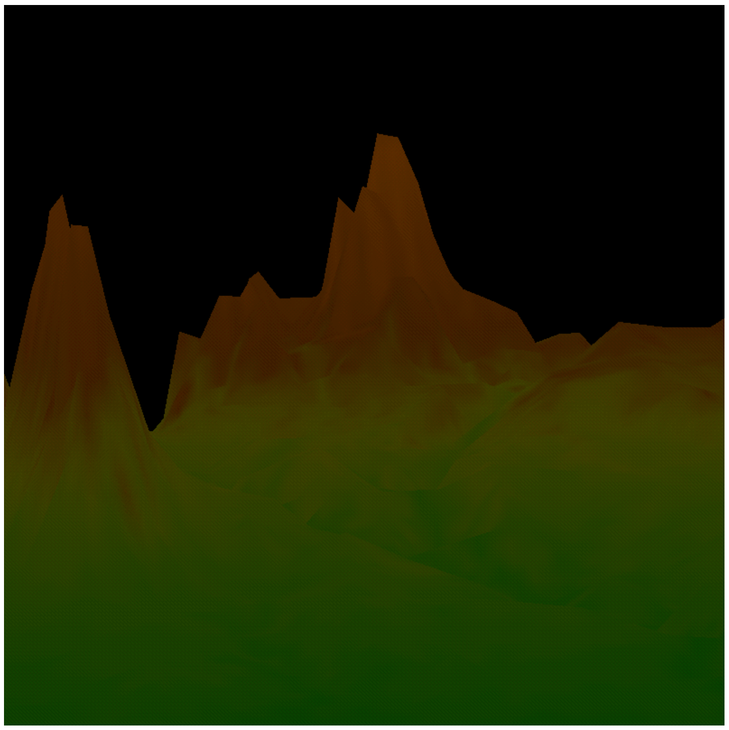

Figure 4.4: Clustered final mesh with texture.

4.2 A 3D Example

We applied the RUSTiC algorithm to a 3D data set produced with Mark Duchaineau's Catmull-Clark subdivision surface generator. This generator produced 96 "patches" of surface coordinates that were then used as input for RUSTiC. The surface texture is pre-shaded using a red light, blue light, and green light on the three axes. Figures 4.5 through 4.8 show a ROAM-generated ROAM triangulation and the RUSTiC-clustered final mesh.

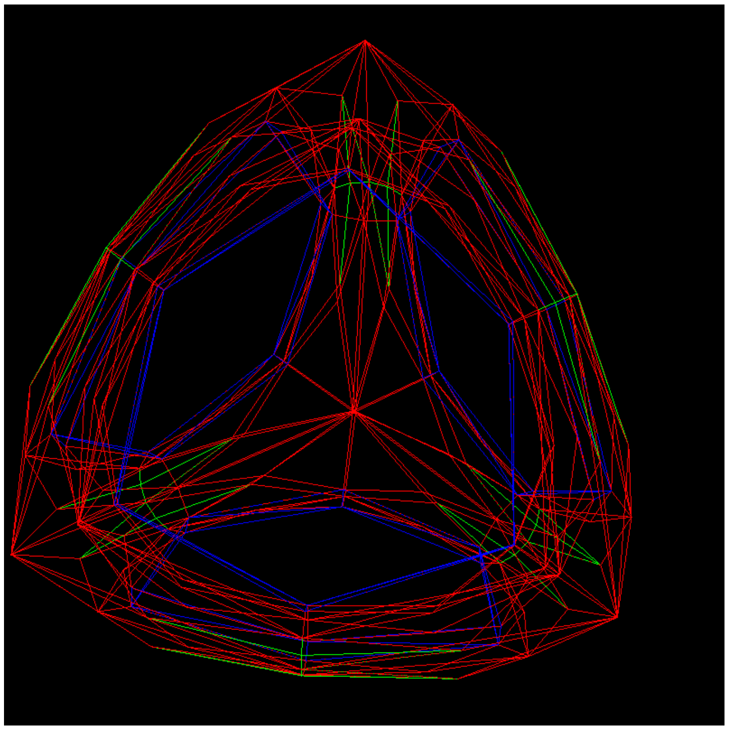

Figure 4.5: A 3D Catmull-Clark subdivision surface ROAM triangulation.

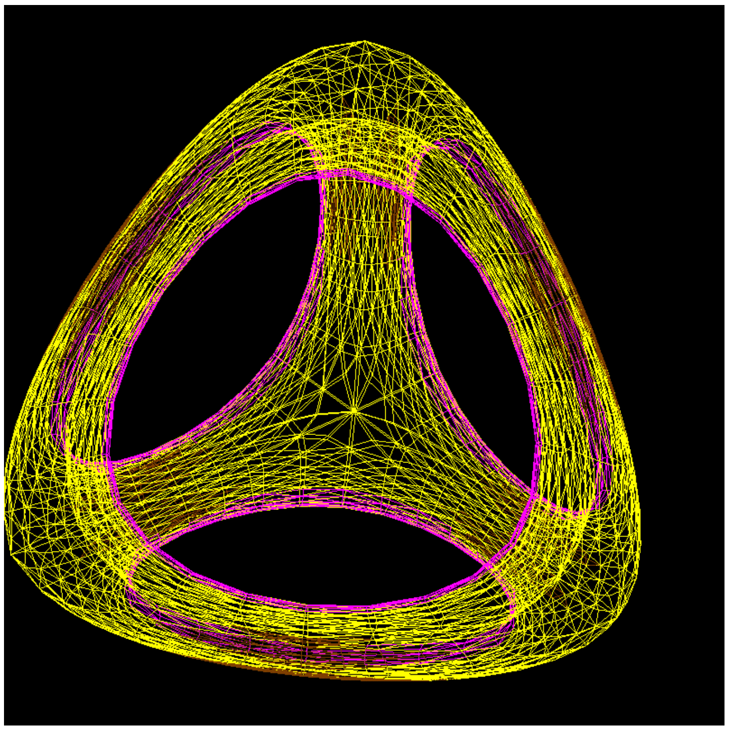
Figure 4.6: Clustered final mesh for shape shown in Figure 4.5.

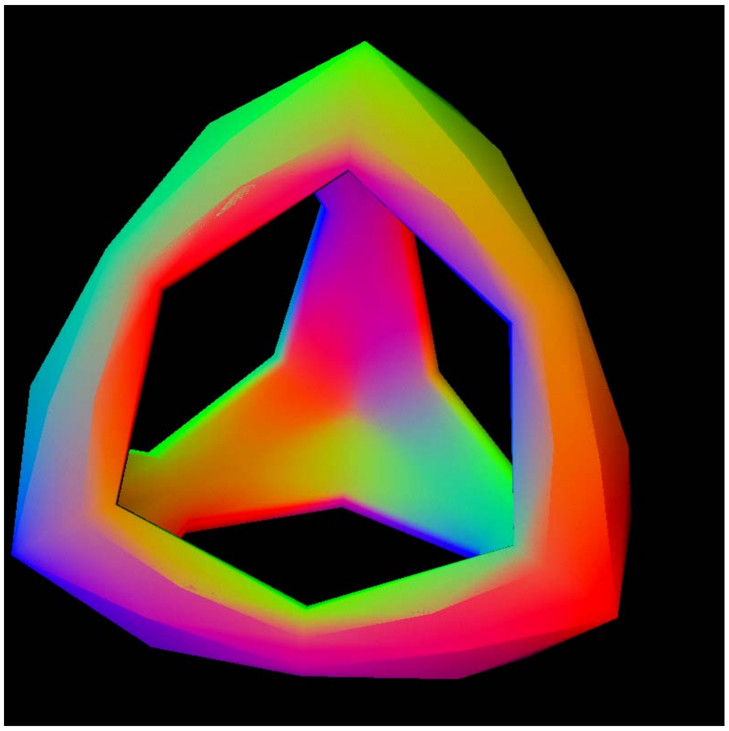

Figure 4.7: Base mesh with texture.

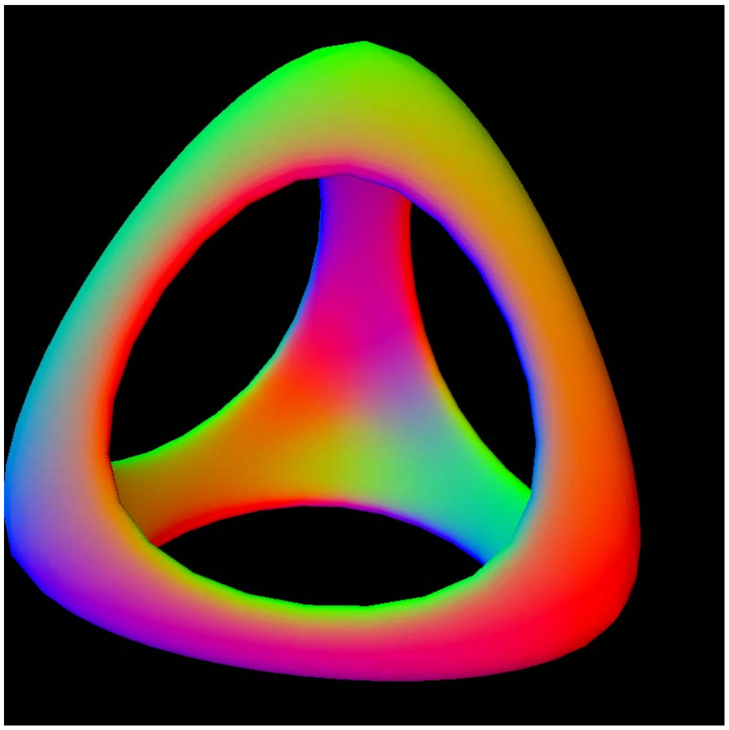

Figure 4.8: Clustered final mesh with texture.

Chapter 5

Conclusion

The RUSTiC method is an improvement of the earlier ROAM technique and increases the polygon count substantially, leading to higher-quality images and increased accuracy. Disadvantages include increased algorithmic complexity, increased memory usage, and more pre-processing steps.

An important part of RUSTiC is choosing an appropriate cluster size. Given the same final triangle count, larger clusters will yield less accuracy than smaller clusters due to the edge constraint. However, larger clusters mean more polygons will be output, increasing overall accuracy.

The original ROAM algorithm is a case of RUSTiC with a target cluster size of $N = 1$. We note that there is not much adaptivity possible when the target cluster size is less than 32 triangles per cluster, due to the edge constraint.

5.1 Implementation

The RUSTiC algorithm was developed at the Center for Applied Scientific Computing (CASC), Lawrence Livermore National Laboratory, and at the Center for Image Processing and Integrated Computing (CIPIC), University of California at Davis. Implementation and testing was done on a Silicon Graphics O2 platform and a 4-processor Onyx2 with an InfiniteReality2 graphics board running the Irix operating system. All programs were written from scratch using object-oriented C++, and the graphics routines were implemented using OpenGL. The testing application is interactive, allowing the user to rotate and zoom through a data set in real time. The Roswell data set was provided by the United States Geographical Survey (http://www.usgs.gov). The 3D Catmull-Clark subdivision surface was provided by Mark Duchaineau at Lawrence Livermore National Laboratory.

5.2 Future Work

One problem with the current ROAM bintree-split method is that it touches a large amount of scattered memory. Each possible triangle in the bintree is allocated individually, and thus individual triangles can end up scattered throughout free memory. Using an efficient queuing method to create the ROAM triangulation may not add much overhead, but it can further scatter data across memory and makes data access less coherent. Unfortunately, this leads to a large amount of cache inefficiency. Future work will involve optimizing the ROAM bintree-splitter to enhance cache coherency. Preliminary research indicates that the total amount of memory touched can be reduced by at least 50 times and access made much more coherent. Using a cache- optimized bintree split can yield incredible speedups on current hardware: optimizing 10K triangles using the queue method took 65 msec, while using a cache-optimized splitter took less than one msec. RUSTiC does not need to be modified to take advantage of such optimizations because they only affect the ROAM triangulation creation.

Another area of future work will change ROAM from a “triangle-based” algorithm into a “diamond-based” algorithm. Preliminary ideas indicate that, by using a diamond-based structure, one can cut per-triangle storage by over 70%. This will lead to a faster algorithm and better cachability. RUSTiC is compatible with such a structure change since the adaptive merge method is set up to work with diamonds.

Bibliography

[1] H. Hoppe A. Lee, H. Moreton. Displaced subdivision surfaces. In *SIGGRAPH '00 Proc.*, 2000.

[2] James H. Clark. Hierarchical geometric models for visible surface algorithms. *CACM*, 19(10):547–554, October 1976.

[3] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification envelopes. In *SIGGRAPH '96 Proc.*, pages 119–128, August 1996.

[4] Daniel Cohen-Or and Yishay Levanoni. Temporal continuity of levels of detail in delaunay triangulated terrain. In *Proc. Visualization '96*, pages 37–42. IEEE Comput. Soc. Press, 1996.

[5] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In *SIGGRAPH '95 Proc.*, pages 173–182, August 1995.

[6] Francine Evans, Steven Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. In *Proc. Visualization '96*, pages 319–326. IEEE Comput. Soc. Press, 1996.

[7] Robert J. Fowler and James J. Little. Automatic extraction of irregular network digital terrain models. *Computer Graphics (SIGGRAPH '79 Proc.)*, 13(2):199–207, August 1979.

[8] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (SIGGRAPH '93 Proc.)*, pages 247–254, 1993.

[9] Paul S. Heckbert and Michael Garland. Multiresolution modeling for fast rendering. In *Proc. Graphics Interface '94*, pages 43–50, Banff, Canada, May 1994. Canadian Inf. Proc. Soc.

[10] Hughes Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *IEEE Visualization 9́8*, pages 35–42. IEEE, October 1998.

[11] Hugues Hoppe. Progressive meshes. In *SIGGRAPH '96 Proc.*, pages 99–108, August 1996.

[12] James T. Kajiya. New techniques for ray tracing procedurally defined objects. *Computer Graphics (SIGGRAPH '83 Proc.)*, 17(3):91–102, July 1983.

[13] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96 Proc.*, pages 109–118, August 1996.

[14] Mark C. Miller. *Multiscale Compression of Digital Terrain Data to meet Real Time Rendering Rate Constraints*. PhD thesis, University of California, Davis, 1995.

[15] Ramon E. Moore. *Methods and Applications of Interval Analysis*. SIAM, Philadelphia, 1979.

[16] Ph. Slusallek H.-P. Seidel" "S. Rottger, W. Heidrich. Real-time generation of continuous

levels of detail for height fields. In *Proc. 6th International Conference in Central Europe on Computer Graphics and Visualization '98*, pages 315–322, 1998.

[17] Hanan Samet. *Applications of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[18] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[19] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *Computer Graphics (SIGGRAPH '92 Proc.)*, 26(2):65–70, July 1992.

[20] Cláudio T. Silva, Joseph S. B. Mitchell, and Arie E. Kaufman. Automatic generation of triangular irregular networks using greedy cuts. In *Proc. Visualization '95*. IEEE Comput. Soc. Press, 1995.

[21] Oliver Staadt, Markus Gross, and R. Gatti. Fast multiresolution surface meshing. In *Proc. Visualization '95*, pages 135–142. IEEE Comput. Soc. Press, July 1995.

[22] Lee R. Willis, Michael T. Jones, and Jenny Zhao. A method for continuous adaptive terrain. In *Proc. IMAGE VII Conference*, June 1996.

[23] J. Xia, J. El-Sana, and A. Varshney. Adaptive real-time level-of-detail-based rendering for polygonal models. *IEEE Trans. on Visualization and Computer Graphics*, 3(2), 1997.

[24] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In *Proc. Visualization '96*, pages 327–334. IEEE Comput. Soc. Press, 1996.