(15)

Optimized Geometry Compression for Real-time Rendering

by

Mike M. Chow

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

Author .....
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by .....
Seth J. Teller
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Optimized Geometry Compression for Real-Time Rendering

by

Mike M. Chow

Submitted to the Department of Electrical Engineering and Computer Science

May 23, 1997

in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Most existing visualization applications use 3D polygonal geometry as their basic rendering primitive. As users demand more complex datasets, the memory requirements for storing and retrieving large 3D models are becoming excessive. In addition, current 3D rendering hardware is facing a large memory bus bandwidth bottleneck at the processor to graphics pipeline interface. Rendering 1 million triangles with 24 bytes per triangle at 30Hz requires as much as 720 MB/sec memory bus bandwidth. This transfer rate is well beyond that of current low-cost graphics systems. A solution is to compress the static 3D geometry as an off-line pre-process. Then, only the compressed geometry needs to be stored in main memory and sent to the graphics pipeline for real-time decompression and rendering.

This thesis presents several new techniques for lossy compression of 3D geometry that compress surfaces 2 to 3 times better than existing methods. We first introduce several *meshifying* algorithms which efficiently encode the original geometry as *generalized triangle meshes*. This encoding allows most mesh vertices to be reused when forming new triangles. The second contribution allows various parts of a geometric model to be compressed with different precision depending on the level of detail present. Together, our meshifying algorithms and the variable compression method achieve compression ratios of between 10 and 15 to one. Our experimental results show a dramatically lower memory bandwidth required for real-time visualization of complex datasets.

Thesis Supervisor: Seth J. Teller
Title: Assistant Professor of Computer Science and Engineering

Acknowledgments

First, I would like to thank my advisor, Seth Teller, for his advice and discussions throughout all stages of my research. Second, I would like to thank many people at Sun Microsystems: Dan Petersen for writing the underlying OpenGL geometry compressor extension and for help on this thesis, Michael Deering and Aaron Wynn for clarifying many compression details, and Scott Nelson for answering memory bandwidth questions and for comments on this thesis. At MIT, thanks go to Marek Techimann for the discussions of the algorithms and help on this thesis. Thanks go to Cristina Hristea for much help and support throughout this arduous journey. My officemates, George Chou and Rebbeca Xiong, shared with me many fun chats and late night hackings. I would also like to thank MIT for being the most challenging and exciting place I have experienced so far.

The word "meshify" was coined by Michael Deering. The scanned models were from Stanford Graphics Lab, Marching Cubes data from Vtk, and others from Viewpoint DataLabs.

Table of Contents

Chapter 1

Introduction

Large geometric models are widely used in visualization applications such as visualizing complex aerospace models, large automotive CAD datasets, large medical datasets of isosurfaces, architectural walkthroughs, and virtual environments. As a result, much current research has focused on managing these large datasets. Two well known techniques are polygon simplification and visibility culling. The former allows an object to be viewed at different levels of detail depending on the viewing distance from the object [8][16]. Visibility culling removes invisible portions of the model from the drawing loop [16]. However, these techniques do not work well when most of the object is visible *and* its full resolution is desired (e.g. during close-up viewing). In this case, the full detailed geometry must still be sent to the graphics hardware for rendering.

As visualization users demand ever larger and more detailed geometric datasets, real-time graphics hardware is increasingly facing a memory bus bandwidth bottleneck in which the large amount of geometry data cannot be sent fast enough to the graphics pipeline for rendering due to slow memory subsystems [15]. For example, to render a large geometric dataset with 1 million triangles at 30Hz (30M triangles/sec) would require 720MB/sec bus throughput between the processor and graphics pipeline (assuming that the geometry is encoded as optimized triangle strips and that each triangle requires 24 bytes including position and normal data). Such high memory bandwidth requirement may be achievable by high-performance, high-cost graphics workstations, but it is not reason-

able for mid-range and low-end machines [3]. Current low to mid-range machines have a memory bus bandwidth of only 250 to 320MB/sec [15][13].

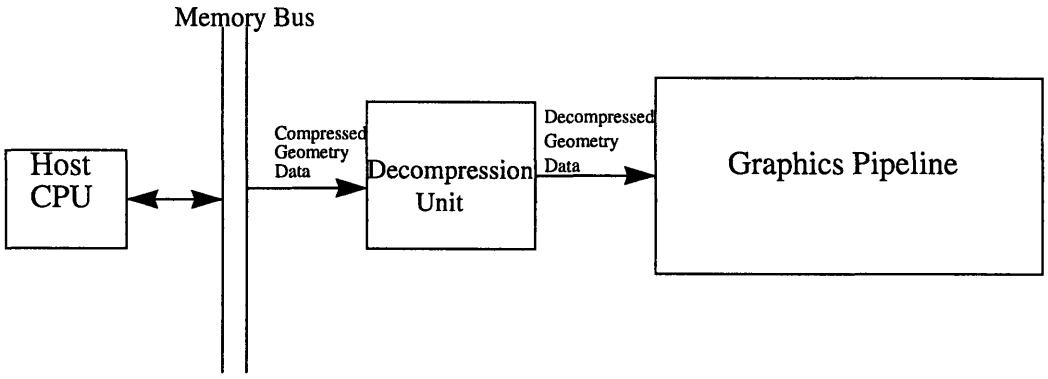


Figure 1.1: Rendering with compressed geometry reduces memory bus bandwidth. Geometry data stored in compressed form is sent across the memory bus to the Decompression Unit, which decompresses the geometry and delegates to the graphics pipeline for rendering.

One proposed solution is to compress the static geometry as an off-line pre-process [4]. Then, the geometry data can be stored in main memory in compressed format. Upon rendering, the compressed geometry is sent to the rendering hardware for real-time decompression using an efficient hardware decompressor [4] and the uncompressed geometry is then rendered. This trades a reduced bus bandwidth for an increased processing load on the graphics pipeline. The flow of the geometry data is shown in Figure 1.1.

Geometry compression is an attractive solution since many of the geometric datasets used by visualization applications are static and do not change from frame to frame. A medical dataset of the human body, architectures and buildings, the engines and chassis of a Boeing 777 are some examples. Using compressed geometry, we can make high-performance graphics much more accessible for low-cost graphics hardware implementations. Based on experimental results of this thesis, with an optimized compression ratio of 10:1, for the same 1 million triangles, we only need to store roughly 3.8 MB of compressed

geometry in main memory and require less than 120 MB/sec memory bus bandwidth at 30Hz. Thus, given a sufficiently fast graphics pipeline, rendering 30 million triangles per second in real-time on a mid-range machine would not be unreasonable in terms of the memory bandwidth requirements (current pixel fill and custom rendering ASICs are also performing at a rapidly rising rates [5]).

The geometry compression techniques described here compress 3D surfaces 2 to 3 times better (i.e. less bytes) than the existing technique [4]. Using the optimized compression results from this thesis, we can reduce the total size of run-time geometry by 10 to 15 times over uncompressed geometry with little loss in image quality. The first optimization efficiently converts the original geometry into the generalized triangle mesh format which allows most vertices to be reused when forming new triangles. We first present two fast local algorithms for building generalized triangle meshes. Then, we show how a global analysis of the mesh can yield better results. Given that geometry compression is an off-line process, we also present a meshifying algorithm that trades execution time for compression ratio. This allows near optimal generalized triangle meshes to be found for many models.

The second optimization allows different regions of a geometric model to be compressed with different precision depending on the level of detail present at each region. This lowers the average bits per triangle for a given object. We also present a method for automating the selection of quantization levels given an user error threshold. This allows unsupervised compression of large geometric databases.

This thesis is organized as follows: Chapter 2 gives an introduction to geometry compression and compares different methods for compressing geometry. Chapter 3 discusses

ways to find generalized triangle meshes and gives several meshifying algorithms. Chapter 4 presents an automatic way to find the right quantization level of an object. It also introduces the variable compression idea. Finally, this chapter gives a summary of the new compression stages. Chapter 5 shows the results of using the optimizations discussed in this work. Chapter 6 explains ways to improve on the existing algorithms and ways to interface with simplification algorithms. Chapter 7 concludes this thesis. This work can be found online at http://graphics.lcs.mit.edu/~mchow.

Chapter 2

Previous Work

2.1 Geometry Compression Concepts

The concept of 3D geometry compression was first introduced by Deering [4]. His compression method is lossy and allows for compression ratio vs. quality trade-offs. First, 3D geometry is represented as a *generalized triangle mesh*, a data structure that efficiently encodes both position and connectivity of the geometry and allows most of the mesh vertices to be reused.

Next, the positions, normals, and colors are quantized to less than 16 bits. This is a lossy step and the final desired image quality determines the quantization level used. These quantized positions and colors are delta encoded. Normals are delta encoded by mapping onto a unit sphere for a table-lookup based encoding. These position and normal deltas are Huffman encoded to output the final compression stream.

This compression method reduces the storage per triangle to between 35.8 to 58.7 bits rather than 192 bits for 3 floating point positions and normals. However, since no efficient methods for decomposing a model into generalized triangle meshes existed at that time, the final compression ratio was limited to between 5 and 7 to one (without much degradation in image quality). Also, that compression method required users to manually experiment with quantization levels given a desired image quality. This trial and error process can be very time consuming for compressing large geometric databases.

2.2 Another Compression Technique

Later, Taubin and Rossignac [18] gave an efficient method for compressing geometry connectivity. Their method decomposes a mesh into spanning trees of triangles and vertices. These trees are encoded separately so that both connectivity and position data are compressed well. They were able to reduce connectivity information to 2 bits per triangle.

However, one disadvantage of this method is that the decompression stage is complicated by large memory requirements and is not amenable to cost-effective hardware implementations. Since decompression requires random access to an array of vertices for a given mesh, a large on-chip cache memory is required to store the referenced vertices. Also, two passes must be made to retrieve the final triangles. One pass must be made for decompression of positions and normals for the vertices. Then, another pass is needed to extract the final triangles which reference those vertices.

2.3 Real-time Decompression Requirements

These factors make it unclear how fast and cost-effective decompression hardware can be built using this compression method. A software decompressor will not meet the stringent real-time requirements of a fast graphics pipeline. In contrast, since Deering's method outputs a linear stream of compressed positions and normals, a high speed and cost-effective decompression unit has been proposed [4] which incrementally extracts triangles from the compression stream for output to the rendering pipeline. Because of the real-time decompression constraints, we chose to focus primarily upon Deering's compression method.

Chapter 3

Efficient Meshifying Algorithms

3.1 Design of Good Generalized Triangle Meshes

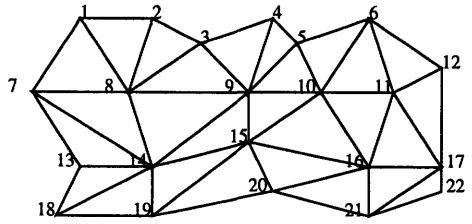
In this chapter, we present our first optimized compression technique: decomposing a given polygonal mesh into generalized triangle meshes which efficiently encode the original geometry.

3.1.1 Generalized Triangle Mesh

A generalized triangle mesh is a compact representation of 3D geometry first introduced by Deering [4]. It consists of a generalized triangle strip [3] with a k-entry fifo to store old vertices (called the "mesh buffer"). Old vertices are pushed onto the fifo and referenced in the future when they are needed again. Thus, instead of storing an entire x, y, z position along with the Nx, Ny, Nz normal components, we only store the fifo index bits representing the old vertex in the fifo. A two-bit vertex header is also needed to specify a vertex.

An example of a generalized triangle mesh is shown in Figure 3.1. Although the geometry can be specified using one generalized triangle strip, many of the interior vertices will be specified twice in the strip. With the help of a mesh buffer to store the old vertices, a generalized triangle mesh efficiently reuses many old vertices by specifying mesh buffer

references instead of repeating those vertices. There is a small cost for each mesh buffer



Generalized Triangle Mesh:
R7, O1,O8p, O2, O3, M9p, O4, O5, M10p, O6, O11p, O12, O17p, O16p, M-4, O15p, O-6, O14p, O-8, O7, M13p, M18, M19, M-3, O20, O-4, O21, O-5, O22

Legend:
R=Restart, O=Replace Oldest, M=Replace Middle
p=push into mesh buffer
Om reads "Replace Oldest on the mth vertex",
O-m reads "Replace Oldest and use the mth entry in mesh buffer"

Figure 3.1: Example of a Generalized Triangle Mesh.

reference due to the bits used to index the buffer entries. However, this thesis assumes a maximum of 4-bit index, yielding a 16-entry mesh buffer (due to hardware limitations). Studies by Evans et al. [7] had also shown that there is little improvement in vertex reuse from very large buffer sizes.

3.1.2 Design of Good Meshifiers

To our knowledge, no general *meshifying* algorithms have yet been proposed for decomposing an arbitrary polygonal mesh into generalized triangle meshes. A general meshifying algorithm would decompose an arbitrary mesh into generalized triangle meshes which contain nontrivial reuse of old vertices. Although there has been research on finding triangle strips in a polygonal mesh [7], converting triangle strips to generalized triangle meshes is a nontrivial task. This is because both the strip sizes and the order of the strips found by the strips algorithm can dramatically impact the reuse of vertices in the mesh. This will be discussed in-depth in next two sections.

Also, the running time of the general meshifying algorithms should depend on the user-specified compression ratio and quality. This way, a high compression ratio will demand the meshifier to use more sophisticated schemes.

We have made some first attempts at designing efficient algorithms which produce compact generalized triangle meshes for arbitrary polygonal meshes. Before describing the algorithms, we introduce several design goals for all meshifying algorithms (shortened as *meshifiers*) which find generalized triangle meshes:

1. Maximize the reuse of vertices.

2. Minimize small generalized triangle meshes since they tend to have a trivial number of reused vertices.

3. Be robust and independent of mesh complexity and topology.

These design goals will be discussed in-depth in the following sections.

3.2 Local Algorithms

In this section we first present a simple, greedy algorithm which finds generalized triangle meshes by linking together generalized triangle strips. Next, we show how to modify this simple algorithm to achieve better results.

3.2.1 Static Local Algorithm

Our simplest local algorithm (shown in pseudocode in Figure 3.3) constructs a generalized triangle mesh by adding a series of generalized triangle strips, one at a time, to a generalized triangle mesh data structure. The algorithm begins by finding a strip of trian-

gles starting from a polygon at a mesh boundary. If no boundary exists (such as a sphere), we randomly pick a polygon from the mesh as a starting point. Once the first strip is found, we pick the next polygon to start from so that the following strip shares the most number of vertices with the previous strip. This situation is shown in Figure 3.2.

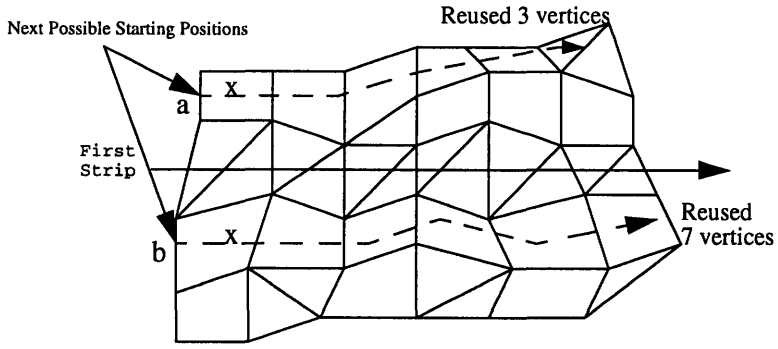


Figure 3.2: Possible starting positions for the next strip.

To maximize reuse of old vertices, the algorithm starts the next strip from a polygon neighboring the start polygon of the previous strip (see 'x' marks in Figure 3.2). Otherwise, starting from another polygon might cause subsequent mesh buffer pushes (on a fifo of 8 vertices, for example) to evict those buffer entries that will be used by the next strip. The pseudocode for the local algorithm is shown in Figure 3.3.

```
Meshify_Local (startFacet, startEdge)
{
prevStrip = NULL;
/* While there are more polygons left in the mesh.*/
while (startFacet exists) do {
   /* Find the a strip from starting facet and edge. */
   triStrip = findTriStrip(startFacet, startEdge, StripLength, &stripFacets);
   /* Assign mesh buffer references to the strip vertices (based on
      prevStrip). Also, mark the reused vertices of prevStrip as mesh buffer
      pushes. */
   assignBufferRefs(triStrip, prevStrip);
   /* Add strip to current generalized triangle mesh. */
   gtmeshAddStrip(triStrip);
   /* Recalculate strip facets' adjacency counts */
   resetFacetAdjacency(stripFacets);
   /* Pick the next starting facet and edge so to maximize
    * reuse of old vertices. */
   pickNextFacet(&startFacet, &startEdge);
   /* Set previous strip to current one. */
   prevStrip = triStrip;
}
```

Figure 3.3: Local Meshifying Algorithm

To find the best starting polygon for the next strip (`pickNextFacet` in Figure 3.3), each possible starting position is scored according to its potential strip's reuse of vertices from the previous strip. The best starting position with the highest potential vertex reuse is chosen for the next strip starting point. We then mark the reused vertices of the previous strip as mesh buffer push vertices (`assignBufferRefs`). Likewise, for the corresponding vertices of the next strip, we assign their mesh buffer references based on their positions in the mesh buffer.

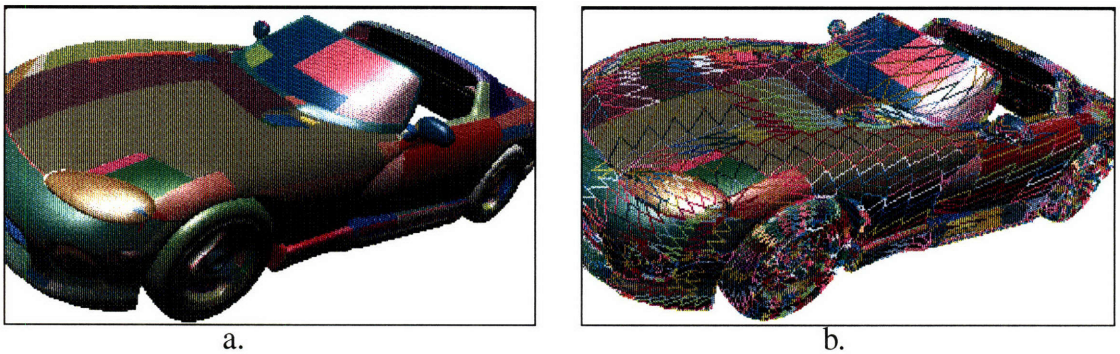
The algorithm terminates whenever it has reached a mesh boundary or there are no more polygons left in the mesh.

After a triangle strip is found, its polygons are no longer considered by the local algorithm, and we remove them from the active mesh polygon list. Next, we update the new mesh boundary created by the removed polygons (`resetFacetAdjacency`). Since removing these strip polygons creates a hole in the mesh (forming a new mesh boundary), we must update the adjacencies of the strip's neighbor polygons. The algorithm then begins the next generalized triangle mesh starting from a mesh boundary polygon. This avoids breaking the mesh into many disjoint pieces and allows better reuse of the mesh vertices.

3.2.2 Discussion and Performance of the Local Algorithm

Since the local algorithm sequentially removes triangle strips from the mesh, and triangle strips are found in linear time proportional to the number of polygons, the local algorithm also has a linear time behavior proportional to the number of input polygons in the mesh. We found this algorithm to be very fast even for very large meshes. Figure 3.4a shows the result after running the local algorithm. Each colored region is one generalized triangle mesh. In Figure 3.4b, the generalized triangle strips are shown in colored wire-

frames. Notice that each strip in a mesh shares a row of vertices with the previous strip and thus are reused. This pattern of vertex reuse is desired in all meshifying algorithms.



a. b.

Figure 3.4: Local Algorithm Example. Left: Each colored region shows one generalized triangle mesh. Right: Each colored wireframe shows one strip of a generalized triangle mesh. Each interior strip reuses a row of vertices from the previous strip.

The *performance* of a meshifier can be measured by the final **vertex to triangle ratio**:

$$r = \frac{totalVerticesVisited - meshBufferReferences}{totalTriangles}$$

totalVerticesVisited is the total vertices in the final generalized triangle meshes.
meshBufferReferences is the number of vertices in the generalized triangle meshes which are references to mesh buffer entries.
totalTriangles is the number of triangles in the mesh.

For example, independent triangles have a 3 to 1 vertex to triangle ratio. (**r** = 3) Generalized triangle strips have a theoretical minimum of 1 vertex per triangle (**r** = 1). In both cases, the number of mesh buffer reference vertices is zero (without using a mesh buffer). With the help of the mesh buffer, however, generalized triangle meshes have a theoretical minimum of 0.5 vertex to triangle ratio (**r** = 0.5) for an infinite, regular mesh grid.

Based on our test results, the local algorithm shows a 0.83 to 0.90 vertex to triangle ratio with an average of 40% vertex reuse.

3.3 Adaptive Local Algorithm

3.3.1 How Strip Length Effects Meshifying

In this section we improve on the previous local algorithm by allowing the lengths of new triangle strips of a generalized triangle mesh to adapt to the local mesh topology. One factor that influences the efficiency of a meshifying algorithm is the length of the triangle strips that are added to a generalized triangle mesh.

Recall that one of our meshifier design goals is to maximize the reuse of old vertices. This demands that the successive triangle strips in our generalized triangle mesh share many vertices. Even if one strip is very long, the successive one may diverge from the previous strip, and most of the old vertices in the previous strip may not be reused. This is shown abstractly in Figure 3.5a.

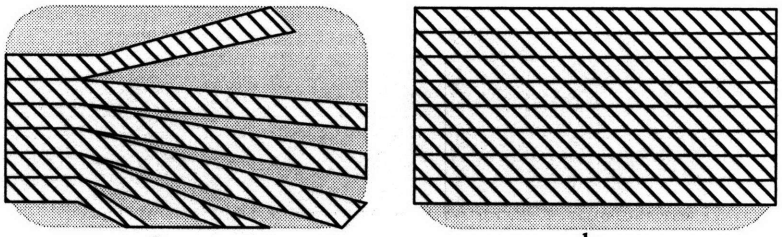


Figure 3.5: How strip size effects a meshifying algorithm. Each striped area represents one triangle strip. In a. long strips lead to diverging strips in general meshes. b. Shows strips on a regular grid mesh. Each strip does not diverge from the previous strip.

One may ask "why can't each successive strip be made to lie exactly on top of the next so that most vertices are shared?" Indeed, this will be case for a regular rectangular grid mesh (see Figure 3.5b). But, for general meshes, it is hard to guarantee the direction of the

strips. Consider the following case where our input mesh contains a "diverging triangle" (see Figure 3.6).

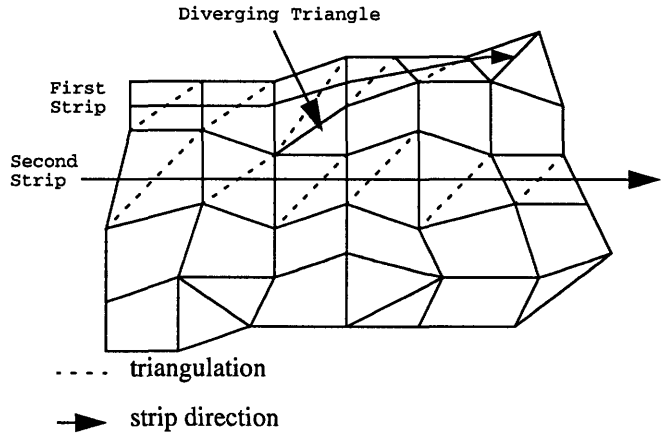


Figure 3.6: A diverging triangle forces first strip to diverge from second strip.

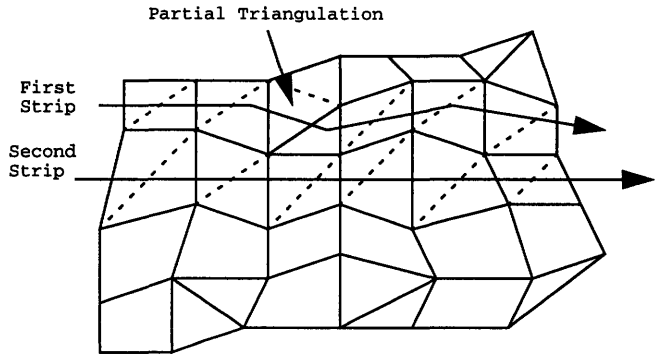


Figure 3.7: Partial Triangulation allows better sharing of vertices between the strips.

This "diverging triangle" forces the direction of the first strip to diverge from the second one. This is because the triangle strips algorithm cannot partially triangulate a polygon to start on a new polygon. This situation might be mitigated if we allow our triangle strips algorithm to do "partial" triangulation as in Evan's [7], where a polygon is allowed to be partially triangulated (see Figure 3.7). This way, a triangle strip's direction has more

freedom. Using partial triangulation, the previous situation will look better (see Figure 3.7). Here, the first triangle strip is forced to share most of its vertices with the following strip. In this work, however, the triangle strips algorithm does not yet do partial triangulation. It will be very useful to add this feature in the future.

It is still unclear whether there is a general solution that prevents two successive triangle strips from diverging. The problem comes from how the mesh was tessellated in the first place. In many cases, 3D modelers output simplified (i.e. polygon-reduced) meshes in which there is no obvious way to find long strips. One possible solution proposed by the author is to "retile" the original points in the mesh and retriangulate such that a meshifying algorithm will easily find successive triangle strips which reuse vertices. This will be discussed later in the Extensions and Future Work Chapter.

In the current work, the method that was adopted to handle "diverging triangle strips" is to first do a "test run" of the meshifying process using our fast local algorithm and find the average number of vertices reused per triangle strip or **Av** for short (See Figure 3.8). Based on this number, we then limit the strip length of our triangle strips algorithm to be **2*(Av - 1)** (since there are roughly twice the number of triangles in a strip than a row of strip vertices). This way, we prevent most of the triangle strips from diverging from the successive ones (see Figure 3.9).

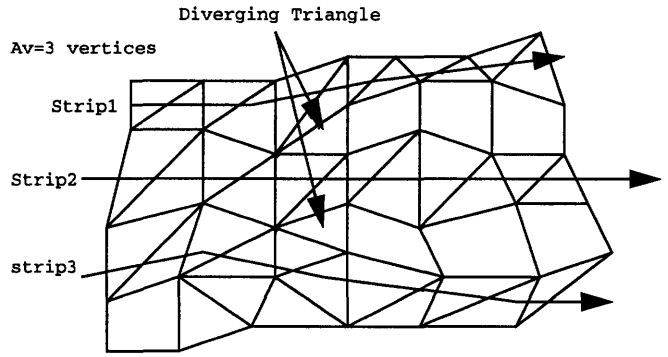


Figure 3.8: A "test run" finds the average reused vertices, **Av**, to be 3. This will set the max strip length.

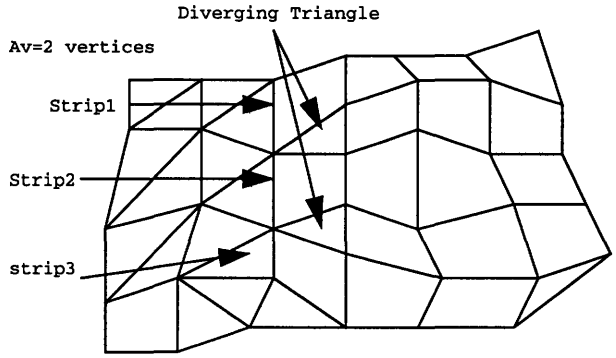


Figure 3.9: With the average number of reused vertices, **Av**, set to 3, the max strip length is now 2*(3-1)=4; this prevents strips from diverging from the successive strips.

This "test run" method of finding **Av** was used by the local algorithm. However, this number, **Av**, is at best only the "average" number of reused vertices per strip. Many regions of a mesh can exhibit far more reused vertices. The static method of limiting the length of triangle strips based on **Av** cannot take advantage of this fact.

3.3.2 Adaptive Local Algorithm

To improve on the local algorithm, one simple modification is to vary the triangle strip length locally for a given portion of the mesh. First, for a given portion of the mesh, we look a given number of strips ahead of the current strip to obtain the average local vertex reuse per strip, **Av'**. Next, we change the strip length limit to the local value **2*(Av' - 1)**. Then, any subsequent strips will have at most **2*(Av'-1)** triangles long. This is shown abstractly in Figure 3.10. The pseudocode for this new algorithm is shown in Figure 3.11. The function stripLengthLookahead looks a given number of strips from the current one and returns a new local strip length (`StripLength`).

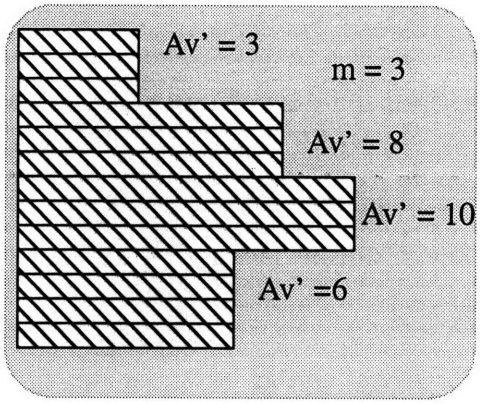


Figure 3.10: By looking ahead to the number of reused vertices, we obtain the average local vertex reuse, Av', for different portions of the mesh. This dynamically changes the local strip length. The lookahead is done every m=3 strips added.

```
Meshify_Local_Adaptive (startFacet, startEdge)
{
prevStrip = NULL;
/* While there are more polygons left in the mesh. */
while (startFacet exists) do {
   /* Lookahead the current strip to find the new, local
    * strip length (StripLength).
    * Do this every StripLookaheadFrequency */
   if ((cnt % StripLookaheadFrequency) == 0) {
         stripLengthLookahead(startFacet, startEdge,
                              StripLookaheadDistance,
                              prevStrip,
                              &StripLength);
   /* Find the strip from starting facet and edge. */
   triStrip = findTriStrip(startFacet, startEdge, StripLength, &stripFacets);
   /* Assign mesh buffer references to the strip vertices (based on
      prevStrip). Also, mark the reused vertices of prevStrip as mesh buffer
      pushes. */
   assignBufferRefs(triStrip, prevStrip);
   /* Add strip to current generalized triangle mesh. */
   gtmeshAddStrip(triStrip);
   /* Recalculate strip facet's adjacency counts. */
   resetFacetAdjacency(stripFacets);
   /* Pick the next starting facet and edge so to maximize
    * reuse of old vertices. */
   pickNextFacet(&startFacet, &startEdge);
   /* Set previous strip to current one. */
   prevStrip = triStrip;
   cnt++;
 }
}
```

Figure 3.11: Adaptive Local Algorithm

We obtain a new **Av'** by doing another lookahead after every ***m*** strips are added to the generalized triangle mesh (`stripLengthLookahead`). The value ***m*** is our lookahead sampling frequency and is dependent on the total triangles in the mesh (the large the mesh the less we sample and higher the value of ***m***). In doing this lookahead, we incur a small cost

of ***c*(n/m)*** over the entire course of the meshifying process, where ***n/m*** is the number of lookaheads and ***c*** is the small constant cost we incur for each lookahead.

Since this dynamic lookahead approach is more adaptive and tuned to the particular sections of a mesh, the adaptive strip length method is able to improve on average by 10% to 15% over the static strip length method. This improvement costs us only slightly, increasing the running time to ***O(c*n/m + n)*** where n is total number of polygons as before and c and m are described above. This is still linear in the number of polygons.

Figure 3.12a shows the resulting generalized triangle meshes after running the local algorithm using static strip length. In contrast, Figure 3.12b shows how adaptive strip length can reduce the total number of small generalized triangle meshes and make the strip length longer on different parts of the model.

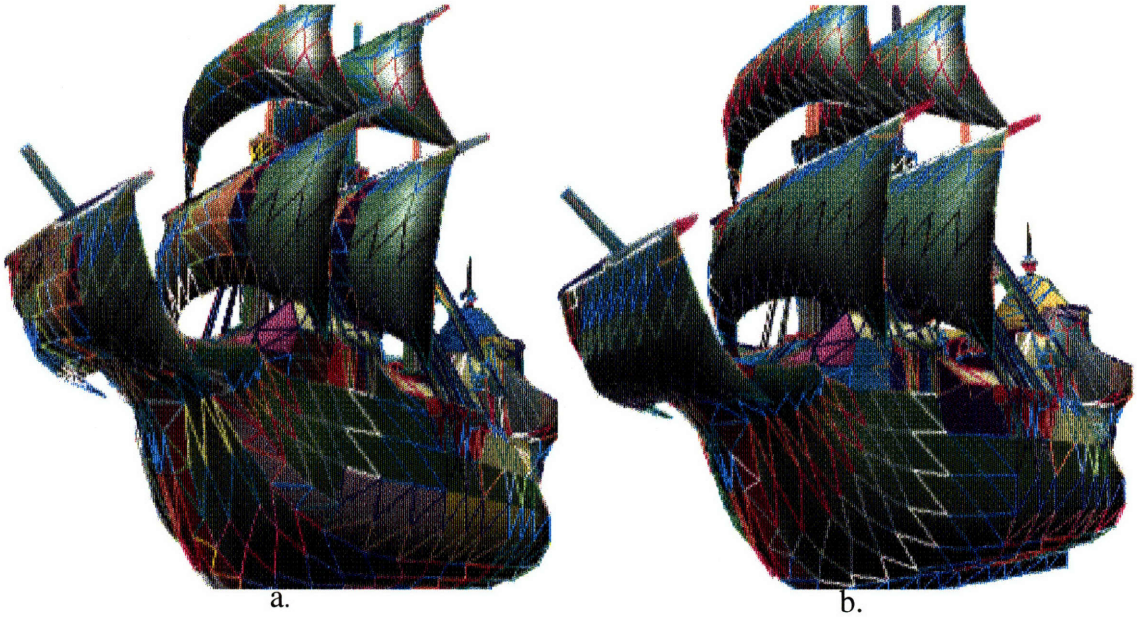


Figure 3.12: Static Strip Length vs. Adaptive Strip Length. In a: A static strip length is found for the entire mesh and is used to limit the lengths of triangle strips. In b: triangle strip length changes adaptively for different portions of the mesh; this allows better reuse of mesh vertices.

3.4 Global Algorithm

While the static local algorithm greedily selects the next triangle strip that maximizes mesh buffer usage, we could also find generalized triangle meshes using a global analysis. Our global method is based on the triangle strips work of Evans et al. [7]. Evans' global algorithm finds long triangle strips by locating large patches of quadrilateral faces in the model.

Typically, 3D CAD modelers using NURBS and spline surfaces output many quads from evaluating splines at sample points. We can exploit this fact by finding large generalized triangle meshes of quads.

The first step of our global algorithm is similar to Evans' algorithm [7] where a large patch of adjacent quads is found. The algorithm begins by choosing a starting quad and "walks" in the *north* and *south* directions from that quad, marking each quad as we go. At each north and south quad, we also "walk" along the *east* and *west* directions, also marking each quad we see. In this way, we "flood" a region of polygons consisting of only quads (see Figure 3.13). From this flooded region of quads, we extract the largest block of adjacent quads so we can create our generalized triangle meshes.

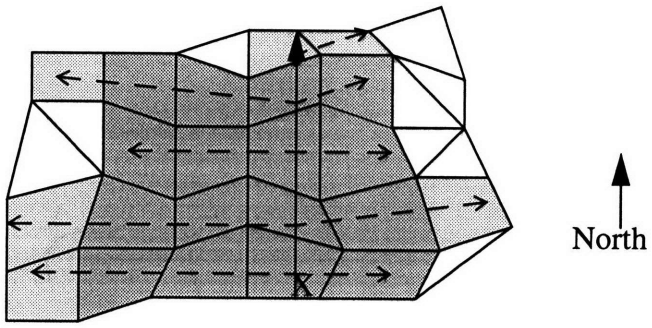


Figure 3.13: Global Algorithm finds a patch of quads. 'X' marks the starting quad. Gray regions show the "flooded" quads. Dark gray regions show the final four sided patch of quads extracted for conversion to generalized triangle mesh.

This block of contiguous quads is made into a four sided region; this will be described below. From this four sided region of quads, we can extract the generalized triangle meshes quite easily. We divide the four sided region into blocks of k by N subregions; each of these subregions will be a generalized triangle mesh whose width is k and height is N (assuming a mesh buffer of k entries). Since the quads in the region are all adjacent, we guarantee that the vertices in each intermediate row of quads are pushed into the mesh buffer and reused by the next strip of triangles.

We extract the largest "rectangular" block of contiguous quads by checking all the quad positions to the north and south directions of the initial quad. At each position we find the largest width and height of adjacent quads. After all positions have been explored, we keep the four sided block with the largest width times height. This is shown as the dark gray region in Figure 3.13.

We combined the global and local algorithms by allowing the global algorithm to first extract the large quad regions of general triangle meshes. Then, we apply our local algorithm using adaptive strip length on the remaining mesh polygons. The worst case running

time of the global algorithm is $O(n^2)$ since extracting the largest four sided block of quads per flooded region can potentially visit a polygon in the region m times where m is the number of polygons in the region. In practice, however, we can greatly accelerate the search for the largest four sided block by setting cut-off thresholds for both the width and the height of a four sided block of quads. The four sided blocks of quads with small widths or heights are automatically rejected by the cut-off thresholds.

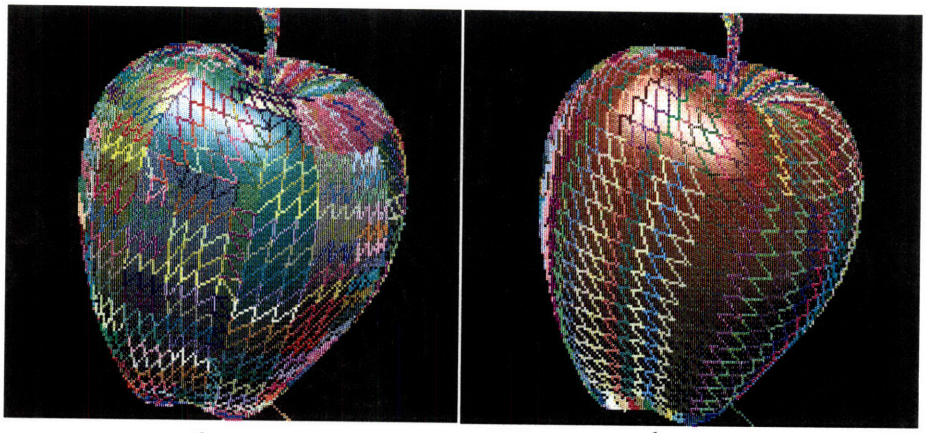
Because the global algorithm is able to examine much larger portions of the mesh first before committing to meshifying, it is able to select large regions that best reuse the mesh vertices. In contrast, the greedy local algorithm can only rely on the next few strips in building a generalized triangle mesh; thus, it often "paints itself into corners".

3.5 A Unified Meshifying Algorithm

Another objective of a good meshifying algorithm is to allow a tradeoff between the compression ratio and the compression time which is dominated by the meshifying algorithm. Since there is no real-time constraint for geometry compression, we would like to allow users to generate near optimal generalized triangle meshes, although at a greater execution time. Our method involves doing a breadth-first search of the possible generalized triangle meshes at each step of the meshifying process. We introduce the idea of a *shadow mesh lookahead* which allows us to look ahead into the space of possible generalized triangle meshes, assign each a score, and choose the one with the maximum vertex reuse.

Before committing to meshify from a starting polygon (selected from a mesh boundary), we first run our best local and global meshifier from that polygon. The resulting "shadow" generalized triangle mesh is scored by its vertex reuse and put aside in a list. Next, we find another starting polygon and apply our best local and global algorithms again to obtain another "shadow" generalized triangle mesh and assign its score. We repeat this M times, where M is a user-specified input. From these M shadow meshes, we select the one that has the highest vertex reuse.

Although, in the worst case, this algorithm runs M times slower than the local algorithm, we are able to approach optimal meshification on many models. Optimal meshification involves finding the minimum number of k by N general triangle meshes, where k is the size of the mesh buffer. In the following example shown in Figure 3.15, the Apple model is meshified using both the static local algorithm (Figure 3.15a) and the shadow mesh lookahead algorithm with $M = 10$ (Figure 3.15b). The lookahead method found two large *16xN* generalized triangle meshes as compared to 17 smaller generalized triangle meshes for the static local algorithm. This lookahead method is able to zero in on the best configuration of general triangle meshes.



a. b.

Figure 3.15: Local Algorithm vs. Unified Algorithm. Left: The colored patches show the generalized triangle meshes found by the static local algorithm. Right: In contrast, the Unified Algorithm found two large generalized triangle meshes. Note the longer strips shown in colored wireframe.

Chapter 4

Variable Compression

In this section, we present our second optimization for compressing 3D geometry: variable compression. We first discuss a method for automatically selecting the quantization threshold for a given model. Then, we present a method for separating a model into variable regions of detail. These regions are later assigned different quantization levels for compression.

4.1 Selecting the Right Quantization

One of the main steps of geometry compression is the lossy quantization of positions where they are truncated to less than 32-bits precision. For many models, quantizations to less than 16 bits results in little loss in quality (the reason will be discussed shortly).

The quantization step involves first normalizing the object's geometry to a unit cube where all x, y, z positions are in the range of [-0.5, 0.5]. There, positions are quantized to q bits ($q <= 16$) by truncating the least significant k bits of the position components, where $k = 16 - q$. This lossy step can be thought of as overlaying a 3D ***quantization grid*** onto the object to be compressed (Figure 4.1a shows this grid in 2D). The 3D quantization grid is divided into units of $1/2^q$ each. Each quantization level corresponds to one quantization grid.

For a given object, if we choose quantization, q, to be too small, the corresponding quantization grid would be too coarse for the given object. Many vertices of the object will

be moved drastically and quantization artifacts will start to appear. This "under-precision" case is shown in Figure 4.1b.

If we pick quantization, q, to be too large, we are in effect overlaying a much denser quantization grid than the object geometry's precision requires. Most of the vertices would remain unchanged after quantization. This "over-precision" case is shown in Figure 4.1c.

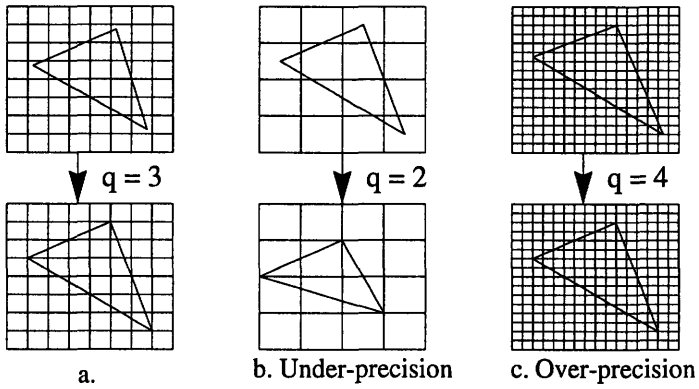
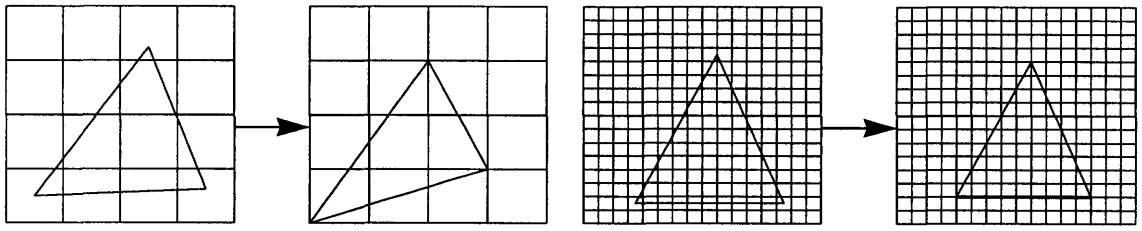


Figure 4.1: Quantization Grids in 2D.

Current geometry compression techniques rely solely on the user to pinpoint the exact value of q to quantize a given object for a desired quality. This is by far a trial and error process and can be quite time consuming when compressing large databases of models. We would like to automate this process by having the program choose the right level of quantization given an user-defined error tolerance.

We have explored a quantization assignment method that first separates the object's geometry into regions of similar triangle sizes (determined by triangle area). This step separates the higher detailed geometry from the lower detailed portions (this step will be discussed shortly). Based on the average edge-length of triangles in each region, we test all possible quantization levels, from q=3 to q=16, to find the best value of q so that its corresponding quantization grid best suits the triangle size.

For each 3D quantization grid of q, we test to see if the triangle size (normalized to unit cube) is within K times a grid unit (length of $1/2^q$). If it is, we use q as our quantization level. The value, K, corresponds to a user-controlled error threshold, $e = 1/K$. It roughly means the range in which the triangle vertices are moved when they are quantized. The smaller the value of e (a larger K), the less vertices shift when they are quantized (K typically varies from 5 to 20). The values of K for two different grid units are shown in Figure 4.2.



If e = 0.25 (K = 4), then q must be 2. If e = 0.1 (K = 10), then q must be 4.

Figure 4.2: Various values of K chosen for a given triangle size. Left: Setting e = 0.25 (K = 4) leads to a larger error in the final quantization (q=2). Right: With e = 0.1 (K = 10), the error is small for the assigned quantization level (q=4).

Using the error threshold, e, we free the user of manually searching for the right level to quantize an object. Our algorithm analytically finds the right quantization level given the error threshold.

4.1.1 Special Considerations for Curved Regions

Although triangle size often gives a good estimate of the number of bits that should be allocated to a particular region, a very curved region can elude this heuristic. When a curved surface is assigned with less bits, quantization artifacts such as "step ladder" effect can arise. This is due to the fact that positions in the region change rapidly along one or

more of the coordinate axes. Thus, the low resolution 3D quantization grid used to approximate this object will have many points clustered in the same grid positions and will be quantized to the same points.

One strategy for avoiding such artifacts is to allocate more bits for the curved regions after the initial quantization is assigned (based on the average triangle sizes). The number of bits to increase is proportional to the region's curvature. This method works well for the majority of the objects and models tested by the author.

4.1.2 Curvature Calculations

The curvature of a region can be found in different ways. One can use the definition of curvature from differential geometry: "the radius of the largest sphere that can be placed on the more curved side of the surface". For a fast estimation, the author has tried two methods, one described by Turk in [17] and one devised by the author which is based on the *largest normal difference*. The second method calculates the maximum normal change between any two vertices in the given region of interest. This maximum normal change is then used as an estimate of curvature. Thus, for a flat region, the maximum normal change is small, and, for a curved region, the maximum normal change is large. This second method was used for its speed and accuracy of approximation.

4.2 Variable Precision Geometry

4.2.1 Separating Mesh into Regions of Different Details

Another related goal we would like to achieve is choosing *different* quantization levels for different parts of an object. The geometry for a given object is rarely described with the same detail everywhere. Often, the high frequency regions of a object are given more detail in terms of smaller triangles. For example, in a 3D model of a person, the geometry describing a person's face may be given more detail than other low frequency regions. For a MCAD model of a car, the two side mirrors with curved surfaces are described with a high density of triangles while the doors with lower density.

Currently, if the user wishes to preserve the highest details in the model, he or she will have to quantize the entire model using the highest quantization even though the highly detailed regions may only account for a small percent of the total geometry. The lower detailed portions of the geometry model are quantized to the same highest quantization level. We call this *static quantization* to distinguish from the *variable quantization* method we are proposing.

For optimal geometry compression, we would like to separate the regions of higher detail from the lower ones. This way, we can preserve the highly detailed regions of a model by quantizing less. In turn, we can quantize more on the less detailed regions. As a result, we can achieve a higher compression since the *average* quantization level for the whole object is lowered (as compared to choosing a static, highest bit quantization).

One natural way to distinguish the levels of detail in a model is by the sizes of triangles (i.e. triangle areas). Our method separates a mesh into different regions based on triangle sizes and then assigns different quantizations to each region. Beginning with a triangulated model, each triangle is successively merged with neighboring triangles to form regions of similar triangle sizes. Two triangles have similar areas if their areas differ

within a certain percentage of the sum of their areas. Next, the small adjacent regions are merged together to reduce total region count (to be discussed shortly). We then assign quantization levels to each region using the assignment method described earlier. These regions are then meshified and compressed.

Since our method of dividing an initial mesh into regions of similar quantization depends only on triangle sizes, this method works on arbitrary meshes of various point densities. It is able to distinguish mesh regions of high detail from those of lower ones. As a result, we can assign more bits to these highly detailed regions. For example, in Figure 4.3, the head model is divided into several regions of high and low details based on triangle sizes. The green and light blue colors on the nose and mouth show high detail and are assigned a 11-bit quantization. The dark blue colored region around the cheeks show lower detail and is assigned 9 bits.

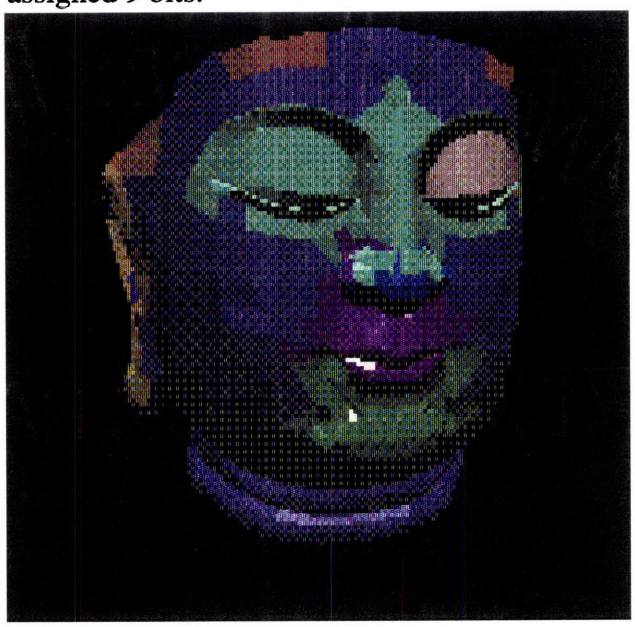


Figure 4.3: Separating regions of similar triangle sizes.

4.2.2 Trade-offs of Separating Mesh Regions

Although it is desirable to describe an object using several quantization levels, we cannot have arbitrarily small regions. Too many regions incur a cost in compression both through many more absolute positions at region boundaries and through lowered vertex reuse by the meshifier. Currently, to prevent cracks at the boundary of two regions with different quantizations, we must stitch together the two regions using absolute precision vertices (rather than relative deltas) at the boundary. Thus, too many regions will cause absolute boundary vertices to dominate the final compression stream.

Another reason to reduce the total regions is to pave the way for the meshifying stage. Having too many regions cuts down the number of triangles per region thus reducing the sizes of the generalized triangle meshes that can be found in the regions. From a general meshifying perspective, the larger a generalized triangle mesh in terms of area, the more vertices will be reused and the lower the final vertex to triangle ratio.

Thus, we have two rivaling objectives: variable compression's need to decompose an object into many regions and meshifying algorithms' need to minimize the region count. The question, then, is which stage of compression yields more significant compression results: variable quantization or meshifying. Variable quantization can save us on the number bits per vertex for each region, but a good meshifier can prevent the redundant vertices from ever appearing in the final compressed stream. A good meshifying algorithm seems to win the contest since it can cause more significant reductions. Hence, we seek to lower the total number of regions.

Our goal then is to divide a mesh into regions of different quantizations *and* to keep the number of regions low. This motivates merging the small regions together. A "small" region is one where the region's triangle count is less than a certain fraction of the total tri-

angles. Also, any two remaining adjacent regions can be merged if they have the same quantization levels and their combined region has that same quantization level.

4.2.3 Summary of Variable Compression

The variable compression method is combined with the rest of compression stages to yield these geometry compression steps:

1. Find regions of similar detail by grouping triangles together based on areas.

2. Merge adjacent regions using the criteria described above, and assign quantization.

3. Meshify and quantize the remaining regions using their calculated quantization.

4. Delta encode the vertices and the normals, and then Huffman encode the deltas to output the final compression stream.

Steps 1, 2, and 3 are new to this work, and step 4 is the same as in previous works [4][18].

Chapter 5

Results

The four meshifying algorithms and the variable compression algorithms described have been implemented as a part of an OpenGL geometry compression extension. The resulting optimized geometry compressor has been tried on over 120 models from several datasets: large medical isosurfaces produced from the Marching Cubes algorithm, range scanned 3D models, mechanical CAD models, and a database of Viewpoint DataLabs 3D models. A sample of these models is presented in Figure 5.3 and Table 5.1.

5.1 The Datasets

5.1.1 CAD and Viewpoint Models

These models were created using 3D modelers. Typically, they consist of a range of primitives, from general polygons to tessellated NURBS. Many of the meshes are results of evaluating NURBS at regular sample intervals to yield the final tessellated objects. As a result, many of these models are dominated by quadrilateral patches. This is useful for global meshifying algorithms that depend on quads for finding large general triangle meshes.

5.1.2 Scanned Meshes

These meshes were 3D scans of objects using a laser ranger finder to evenly sample the depth information of an object. The result is a very fine, sub-millimeter sampling of the original object. This usually results in very dense set of triangles (in the millions).

Because of the current global algorithm's dependence on quads, the scanned meshes are first preprocessed by a "triangle to quads" converter. This tool merges any two adjacent, near-coplanar triangles into a quad (also checking for convexness of the resulting quad). Since the merging is greedy, this runs very fast and is linear in the number of triangles. The resulting models contain a majority of quads with a small number of triangles that cannot be merged. The results of the scanned meshes from Table 5.1 are based on the meshes generated from the **tri2quad** converter taking scanned meshes as inputs.

5.1.3 Medical Datasets

Another source of complex models is from MRI scanned medical datasets. Typically, 2D slices of an object are taken using a MRI scanner. Together, these slices form a sampled 3D object.

To reconstruct the original 3D surface, algorithms such as the Marching Cubes surface construction method [9] are used. The triangles created are very regular across the surface and, often, very dense. The reconstructed surface is often simplified to create an approximate surface. For our testing purposes, the inputs to the compressor are the original reconstructed surfaces. This is also shown in the Table 5.1 (models with the "Marching Cubes" prefix).

5.2 Compression Results

5.2.1 Variable Compression Results

Figure 5.3b and e show two models compressed using the variable region quantization method. In Figure 5.3c and f, each detailed region is assigned a quantization level (one color per level). In contrast to the static quantization method as shown in Figure 5.3a and d, which assigns one quantization to the whole object, our variable quantization preserved the high frequency details (such as the maiden's face in Figure 5.3d) while still retaining a low *average* quantization for the entire model. The large models shown in Figure 5.3h, k, and m were compressed without any manual intervention using our automated quantization assignments. By setting the initial error tolerance, to be small (e = 0.1), the compressed models were virtually indistinguishable from the originals (Figure 5.3g, j, and l).

5.2.2 Compression Statistics

Table 5.1 shows our optimized compression statistics. The last two columns show the compression ratios of uncompressed geometry (in both ASCII and binary formats) to the compressed generalized triangle meshes. We calculated the "original size" columns by counting the bytes for both ASCII file size of the models (with positions, normals, and face indices) and the binary generalized triangle strips with positions and normals represented with full-floating point precision. It is useful to compare with ASCII file sizes since many existing 3D surface file formats are still in ASCII formats (e.g. Wavefront and VRML).

In the "compressed strip bytes" column, we compressed the models using generalized triangle strips without using any meshifying algorithms or variable quantization methods. This is comparable to results using existing compression techniques [4].

Because of the meshifying algorithms and the variable compression method, our results show a two-fold improvement over existing geometry compression techniques and an overall **24** to **36** compression ratio over ASCII bytes and **10** to **15** compression ratio over binary triangle strips bytes. Our variable compression results are shown in "ave quant" column. As compared to a static quantization (pos quant column), a variable quantization method lowered the *average* quantization level for the whole object while still preserving the high frequency details. The values in the "pos quant" column (showing static quantization) were manually chosen so to preserve the highest details in the models. To match and preserve the same details, we picked a low error threshold $e = 0.1$ for our quantization assignment.

Object	# tris	bits/ tri	vert/ tri	pos quant	ave quant	bits/ xyz	norm quant	bits/ norm	original size (ascii)	original size (binary strips)	compressed strip bytes	compressed bytes	ratio over binary	ratio over ascii
Apple	10,556	20.76	0.63	11	9.6	13.3	12	4.2	996,960	420,784	74,705 (5.6X)	27,395	15.4X	36.4X
Dodge Viper	55,759	23.7	0.69	12	10.6	13.6	13	5.5	5,649,794	1,934,996	280,434 (6.9X)	165,665	11.7X	34.1X
Skull	60,339	26.0	0.73	12	10.1	14.9	13	5.7	6,003,428	2,374,316	395,729 (6.0X)	196,415	12.1X	30.6X
Scanned Bunny	69,451	32.2	0.78	13	12.0	13.7	13	5.9	7,062,795	3,070,340	582,250 (5.3X)	279,565	11.0X	25.3X
Hindu Statue	74,172	22.5	0.69	12	9.9	12.6	13	5.6	7,674,068	3,067,736	516,019 (5.9X)	208,941	14.7X	36.7X
Marching Cubes Jaw	75,556	32.7	0.94	13	10.2	12.5	13	4.9	7,651,146	3,442,376	646,379 (5.3X)	308,779	11.1X	25.0X
Turtle	102,998	23.4	0.63	12	11.3	14.4	13	5.7	10,459,787	3,847,592	599,173 (6.4X)	301,866	12.8X	34.7X
Acura Integra	109,576	25.4	0.75	13	10.4	13.5	13	5.6	11,112,993	4,087,496	657,156 (6.2X)	349,178	11.7X	31.8X
Roman Coliseum	135,516	26.9	0.75	12	10.8	14.4	14	5.2	13,866,559	5,374,936	924,406 (5.8X)	455,839	11.8X	30.4X
Schooner Ship	206,732	27.5	0.71	14	11.9	15.6	13	5.1	21,064,503	8,071,098	1,317,302 (6.1X)	711,651	11.3X	29.6X
Marching Cubes Skull	229,539	30.4	0.93	13	11.8	12.2	13	5.2	20,789,349	9,260,328	1,785,591 (5.2X)	872,248	10.6X	23.8X
Scanned Buddha	293,233	30.1	0.94	13	11.3	14.1	13	5.0	29,937,845	12,335,301	2,601,412 (4.7X)	1,104,605	11.2X	27.1X

Table 5.1:

5.2.3 Performance of Meshifying Algorithms

To study the performance of our meshifiers, we tested our four meshifiers: static local, adaptive local, global, and shadow mesh lookahead algorithms on a subset of the objects (using a mesh buffer size of 16 entries). As shown in Figure 5.1, both the adaptive local algorithm and the global algorithm are able to out-perform the simple static local algorithm. Due to its dependence on quads, the global algorithm was only slightly better than the adaptive local algorithm. The shadow mesh lookahead algorithm, with M = 10, was the best among the four algorithms, bringing the **vertex to triangle ratio** closer to the theoretical minimum of 0.5. This was used to produce the "vert/tri" column in Table 5.1. We found the meshifying process to dominate the compression time, which is in the range of 1 to 50 minutes for objects of different complexities and desired compression ratios. Figure 5.3i shows the generalized triangle meshes extracted by the meshifying algorithm.

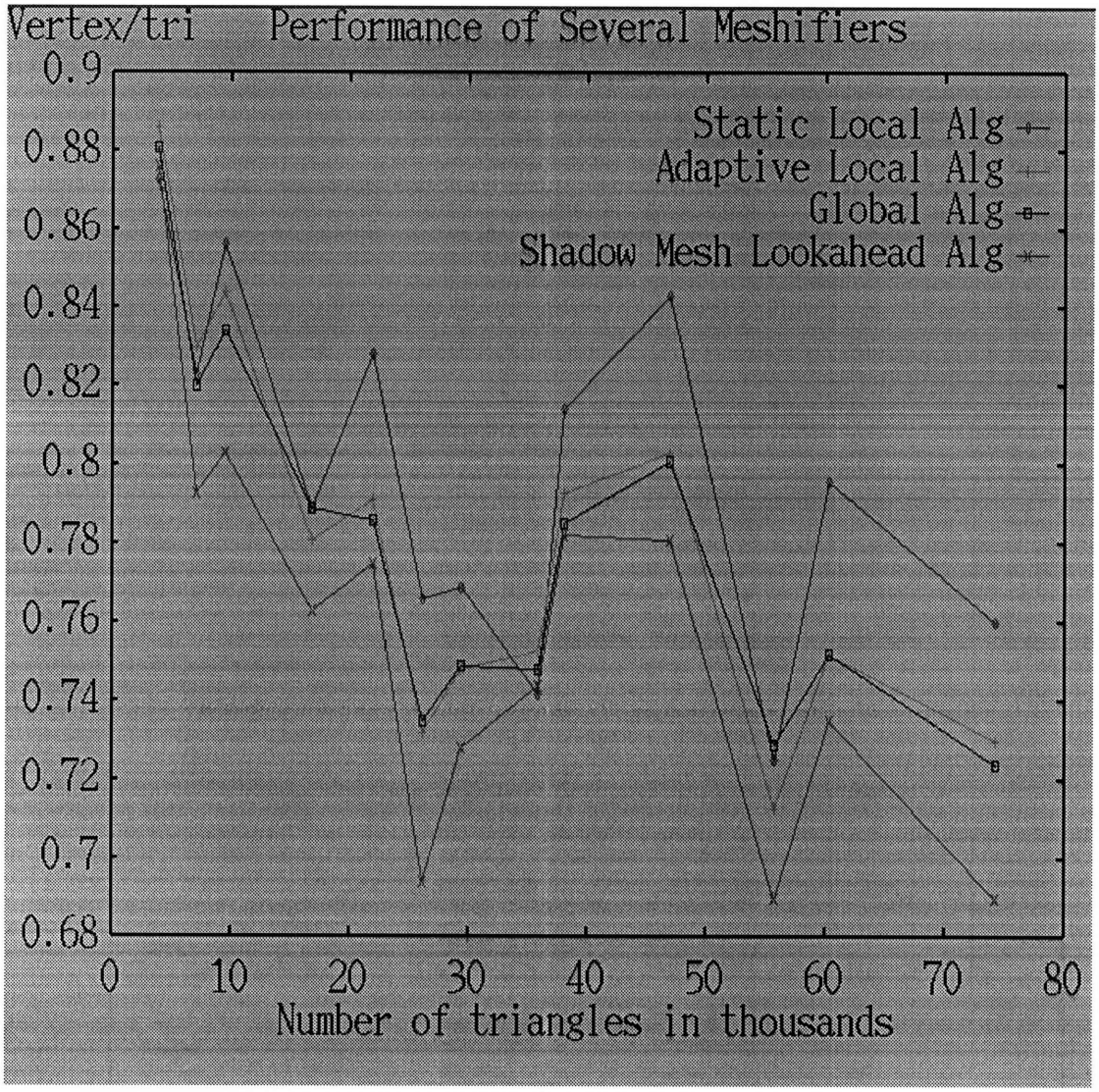


Figure 5.1: Comparisons of Several Meshifying Algorithms. The horizontal axis shows the number of triangles in the input models. The vertical axis shows the **vertex to triangle ratio** of the resulting generalized triangle meshes found by each of the algorithms (the lower this ratio the better the performance). The *shadow mesh lookahead* algorithm performed the best among the four algorithms. The *global* and *adaptive* local algorithms performed about the same. The *static local* algorithm was the worst of the four algorithms.

5.2.4 Real-time Memory Bandwidth Requirements

We also plotted a graph of the real-time memory bus bandwidth requirements for a sample of the tested objects. The graph shown in Figure 5.1 compares the memory cost of rendering uncompressed vs. compressed geometry in real-time (30Hz). The vertical axis shows the memory bus bandwidth (MB/sec) required to send a given amount of geometry memory to the rendering pipeline.

The results from our tests clearly show the advantage of using compressed geometry with respect to memory bandwidth costs. Based on our experimental results, as the scene complexity reaches 1 million triangles, the equivalent memory bus bandwidth required for real-time rendering at 30Hz surpasses 1GB/sec when using uncompressed geometry encoded as triangle strips. This is greater than the ideal 720 MB/sec bandwidth using optimized triangle strips (with 24 bytes per vertex) since the average strip size decreased for larger models. The actual **vertex to triangle ratio** for triangle strips was greater than the ideal 1:1 (about 1.25:1). However, using our optimized geometry compression, the

required bandwidth is reduced to less than 105MB/sec, which is manageable even on existing low-end machines.

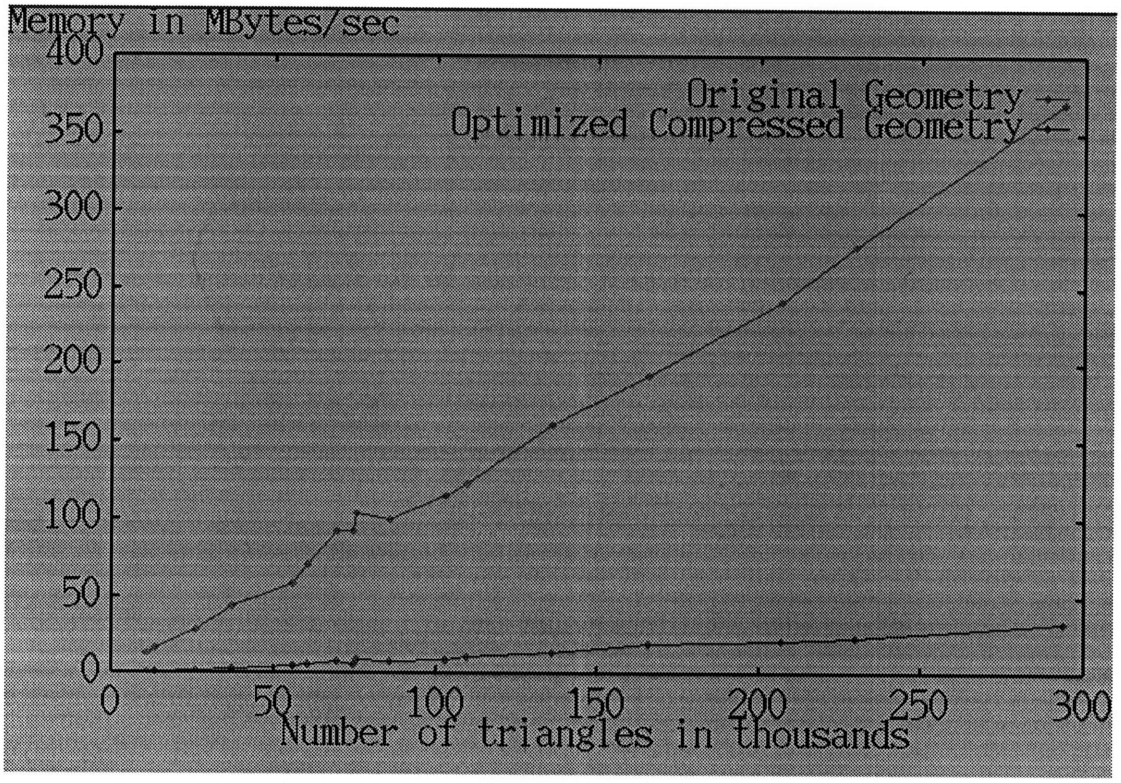
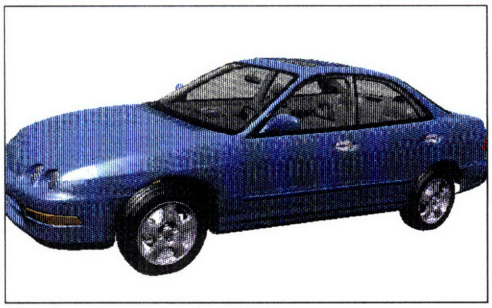
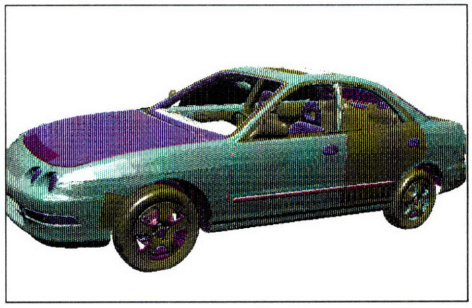


Figure 5.2: Real-time Memory Bandwidth Requirements

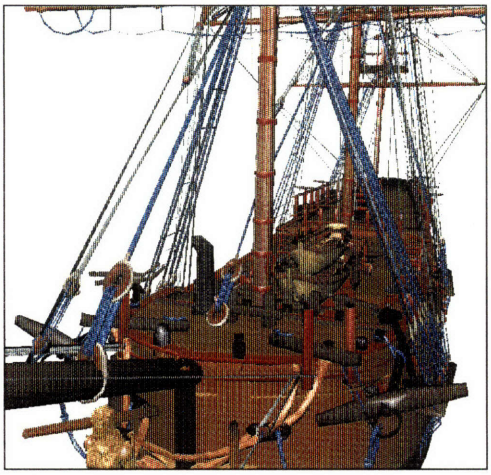
a. Acura, static quantized to 10 bits.



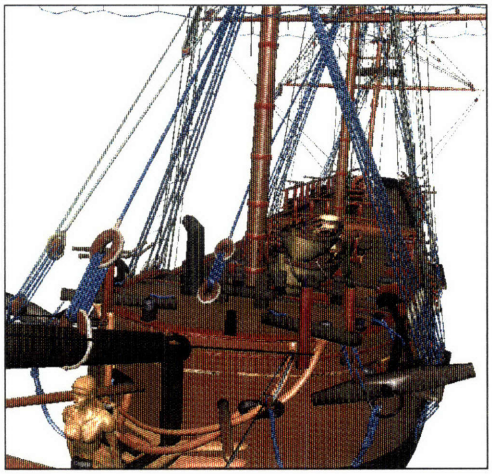
b. Acura, variable quantized to 10.4 bits.



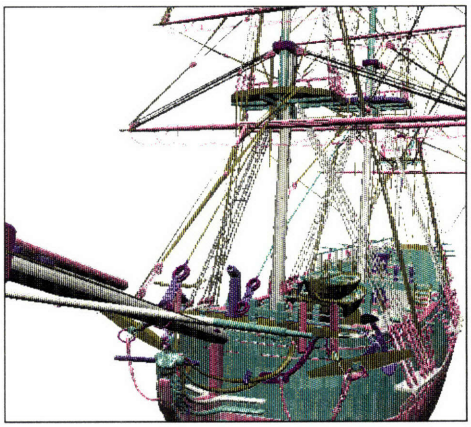
c. Quantization Assignments.



d. Schooner, static quantized to 12 bits.



e. Schooner, variable quantized to 11.9 bits.

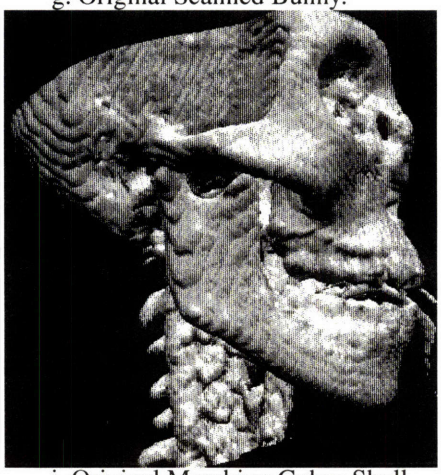


f. Quantization Assignments.

g. Original Scanned Bunny.

h. Quantized to average 12.0 bits.

i. Meshifying Results. Each generalized triangle mesh is shown with one color.



j. Original Marching Cubes Skull.

k. Quantized to average 11.8 bits.

l. Original Buddha. m. 10.1 bits quantization.

igure 5.3. Optimized Compression Results. **a, b** and **e, f** compare static vs. variable quantization. Notice how the small details are pre-rved in **b** and **e** but not in **a** and **d** which were quantized statically. **c** and **f** show the quantization assigned to each region; each color shows ie quantization level. **i** shows the generalized triangle meshes found by the meshifying algorithms. **g-m** compare the original surfaces with ir compressed versions using our quantization assignment method (error = 0.1).

Chapter 6

Extensions and Future Work

6.1 Combining Geometry Simplification with Compression

Geometry compression plays the role of reducing the total memory for storage and for network transmissions. However, geometry models often come in different resolutions and levels of detail. Typically, when a model is viewed from far away, its lower detail version is displayed instead of the full resolution version.

One way to minimize memory requirements for a complex model is to *both* simplify it into levels of detail *and* compress each resolution. Although one can simply simplify a model into different levels of detail and then apply geometry compression to each level, one may ask whether the method of simplification can effect the quality of final compression. Indeed, since the first stage of geometry compression is finding large generalized triangle meshes, the final compression would proceed better if the simplification method easily allows optimal generalized triangle meshes to be found by the meshifying algorithms.

There are literally tens of different geometry simplification methods that have been proposed; each has its strengths and weaknesses. Some notable ones can be found in [2][8][10][12][17]. Since the local meshifying algorithms proposed here depend on finding long triangle strips, a simplification based on local vertex-removals such as the decimation algorithm [12] would break many long triangle strips in the original mesh, causing the average strip size of the final decimated mesh to diminish.

The author has experimented with several simplification algorithms and showed how they can effect the meshifying algorithms. For both decimation [12] and other local algo-

rithms such as simplification envelopes [2], the final triangle strip sizes decreased for successively lower resolutions (for the reason given above).

Polygon Retiling

One simplification algorithm, the polygon retiling method [17], shows significant promise for interfacing with meshifying algorithms. The polygon retiling method first sprinkles on the original surface a smaller number of vertices than the original number of vertices in the mesh. These vertices are randomly distributed. Next, an energy minimization is applied to these points such that points closer to each other are assigned more energy. This induces points to distribute evenly along the surface of the mesh after several iterations of this minimization. These points are then triangulated along the surface to produce the simplified mesh.

Since the polygon retiling method produces evenly sized triangles (i.e. no "sliver" triangles), one can easily trace out, by inspection, long consecutive lines of triangle strips. These long strips might allow the meshifying algorithm to identify very large generalized triangle meshes easily. Also, it's possible to convert any two adjacent, near-coplanar triangles (since they are evenly sized) into quads and thus enabling a global meshifying algorithm to run well on the large quad regions. More work is needed to see if polygon retiling can help interface with geometry compression.

6.2 Multiresolution, View-dependent Simplification with Compressed Geometry

There is a current surge of interest in viewpoint-dependent, dynamic simplifications that allow progressive multiresolutions of a given model to be displayed at run-time [10][8][19]. This improves over a fixed, level-of-detail method where a given number of resolutions of a model are successively used depending on the viewing position. First, the

total storage of multiresolution, progressive meshes is usually smaller than storing all the levels of detail of a model. Second, multiresolution meshes allow small portions of a given model to be simplified rather than swapping the entire model with a lower resolution one in the case of a level-of-detail approach. More advantages and differences with the level-of-detail-based approaches can be found in Hoppe's Progressive Meshes paper [8] and from Rossignac's paper [11].

Compressed generalized triangle meshes can be used as a basic primitive for building multiresolution, progressive mesh representation of a given model. One way is to represent the original model as an hierarchy of generalized triangle meshes of various resolutions. This hierarchy (such as an octree) is computed as preprocess just like compression. Each level stores compressed generalized triangle meshes of different resolutions. At run-time, this hierarchy can be traversed, and the portions of the hierarchy farther from the viewer can be swapped with lower resolution generalized triangle meshes higher up in the hierarchy.

This way, complex geometry models can be decomposed into a hierarchy of multiresolution, compressed meshes. Some portions of a model can be swapped with lower resolution meshes dynamically, at run time. More research will reveal the advantages and shortcomings of this multiresolution method.

6.3 Another Compression Method: Wavelet Compression

The current method of compressing 3D geometry pursued in this work is one of several ways for compressing 3D surfaces. Another actively researched area is using wavelets to both represent 3D surfaces [6] and for compression. The author has experimented with one method for representing 3D surfaces in the wavelet domain (see [1] for details). Using

a signed distance function from a surface, one can represent any 3D surface as a set of sampled distance values. This three-dimensional distance function can be represented in the wavelet domain as a set of scaling functions and wavelets at different resolutions [14]. Using a pyramidal wavelet transform to convert the distance function into a set of wavelet coefficients, one can then apply wavelet compression by truncating the smaller wavelet coefficients. This can yield as much as 150:1 compression ratio of the original surfaces with little loss in surface quality.

There is still much work to be done to make wavelet compression feasible for hardware implementations with high decompression speed requirements. Also, since wavelets are a natural way to represent a function at multiresolutions, it is possible to introduce a multiresolution method for compressing and rendering of surfaces using wavelets.

Chapter 7

Conclusions

This thesis has presented two new techniques for better compressing 3D geometry. We have made the first attempts at designing efficient meshifying algorithms which decompose a given model into the compact generalized triangle mesh format. This representation significantly reduced the number of vertices used to specify a given number of triangles, thus reducing the total geometry size. Two fast, local meshifying algorithms were introduced that construct generalized triangle meshes by adding a sequence of generalized triangle strips that maximize the reuse of previous strip vertices. We showed how a global analysis for finding large regions of quads can be used to extract large generalized triangle meshes. Our unified meshifying algorithm lets users trade execution speed for compression ratio and can approach optimal generalized triangle meshes for some models. We used this algorithm to meshify and compress over 120 models of various complexity with good results.

The second contribution is a method for automating the quantization selection process for geometry compression. This allows an unsupervised compression of large geometric databases to be feasible. We also introduced a simple method for compressing the different regions of a geometric model with variable precision depending on the level of detail present at each region. High detailed regions are preserved by being quantized less, while low detail regions are quantized more. This increased the compression ratio by lowering the *average* quantization level of an object.

Together, our two optimizations compress arbitrary 3D surfaces 2 to 3 times better than an existing compression method based on Deering's work. Our experimental results demonstrated a significant reduction (10 to 15 times less) in the required memory bus bandwidth for real-time rendering of complex geometric datasets. This reduction in memory bandwidth requirements allows high-performance graphics to be accessible for low-cost hardware implementations.

References

[1] Chow, M. and Teichmann, M. A Wavelet-based Multiresolution Polyhedral Object Representation. To appear in SIGGRAPH Visual Proceedings, 1997.

[2] Cohen, J., Varshney A., Manocha D., Turk, G., Weber H., Agarwal, P., Brooks, F., and Wright, W. Simplification Envelopes.*Computer Graphics* (Proc. SIGGRAPH 1996), pages 119-128.

[3] Deering, M., and S. Nelson. Leo: A System for Cost Effective Shaded 3D Graphics. *Computer Graphics* (Proc. SIGGRAPH), pages 101-108, August, 1993.

[4] Deering, M. Geometry Compression. *Computer Graphics* (Proc. SIGGRAPH), pages 13-20, August 1995.

[5] Deering, M., S. Schlapp, M. Lavelle. FBRAM: A new Form of Memory Optimized for 3D Graphics. *Computer Graphic*s (Proc. SIGGRAPH), pages 167-173, August 1994.

[6] Eck, M., T. DeRose, T. Duchamp, H. Hoppe, M. Loundbery, W. Stuetzle. Multiresolution Analysis of Arbitrary Meshes, Proc. SIGGRAPH 95, pages 173-182.

[7] Evans, F., Skiena, S., and Varshney, A. Optimizing Triangle Strips for Fast Rendering. *Proc. Visualization '96*, pages 321-326.

[8] Hoppe, H. Progressive Meshes. *Computer Graphics* (Proc. SIGGRAPH), pages 99-108, August 1995.

[9] Lorensen, W. E., Cline, H. E. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics* (Proc. SIGGRAPH), pages 163-169, 1987.

[10] Luebke, D. and Erikson, C. View-Dependent Simplification of Arbitrary Polygonal Meshes. To appear in *Computer Graphics* (Proc. SIGGRAPH 1997).

[11] Rossignac, J. and Borrel P., Multi-Resolution 3D Approximations for Rendering Complex Scenes, *Modeling in Computer Graphics*, Springer-Verlag, 1993, 455-465.

[12] Schroeder W. J., J. A. Zarge, W. E. Lorensen. Decimation of Triangle Meshes. *Computer Graphics*, 1986.

[13] Silicon Graphics, Inc. Indigo2 IMPACT System Architecture. 1996, *http://www.sgi.com/Products/hardware/Indigo2/products/arch.html*

[14] Strang, G. and Nguyen, T. Wavelets and Filter Banks. Wellesley-Cambridge Press, 1996, Massachusetts.

[15] Sun Microsystems, Inc. The UPA Bus Interconnect. *Ultra1Creator3D Architecture Technical Whitepaper*, 1996, *http://www.sun.com/desktop/products/Ultra1*

[16] Teller, S., and Sequin, C. H. Visibility Preprocessing for Interactive Walkthroughs. *Computer Graphics* (Proc. SIGGRAPH) 1991, pages 61-69.

[17] Turk, Greg. Re-Tiling Polygonal Surfaces. *Computer Graphics* (Proc. SIGGRAPH 1992), pages 55-64.

[18] Taubin, G.and Rossignac, J. Geometry Compression Through Topological Surgery.IBM RC-20304, Jan. 1996.

[19] Xia, J. C. and Varshney, A. Dynamic View-Dependent Simplification for Polygonal Models. *Proc. Visualization '96*, pages 327-334.