A LOCALIZED APPROACH TO IMAGE-DRIVEN SIMPLIFICATION

by

AARON RICHARD PFEIFFER

A THESIS

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 2001

"A Localized Approach to Image-driven Simplification," a thesis prepared by Aaron Richard Pfeiffer in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science. This thesis has been approved and accepted by:

Gary W. Mey

Dr. Gary W. Meyer

6/5/01

Date

Accepted by:

Dean of the Graduate School

An Abstract of the Thesis of

Aaron Richard Pfeiffer for the degree of Master of Science
in the Department of Computer and Information Science
to be taken June 2001

Title: A LOCALIZED APPROACH TO IMAGE-DRIVEN SIMPLIFICATION

Approved: Gary W. Meyer
Dr. Gary W. Meyer

Local calculations are used to extend the notion of image-driven polygon mesh simplification. An implementation of the existing image-driven simplification algorithm is presented and it is shown to have the most important features of the original method. A simple technique is developed to evaluate the collapse cost for an edge by generating and considering only image data local to that edge. In addition, the requirement of maintaining images of the original mesh is eliminated. These techniques are combined with the traditional image-driven simplification approach to enable simplification without at any point rendering the complete mesh. Benefits of this method include automatic prevention of interior simplification, easier sampling camera definition, and more flexibility for future implementations. A technique to reduce texture shifting during mesh simplification, using real-time solid texturing, is also described.

CURRICULUM VITA

NAME OF AUTHOR: Aaron Richard Pfeiffer

PLACE OF BIRTH: Drain, Oregon

DATE OF BIRTH: November 1, 1977

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

Oberlin College
University of Oregon

DEGREES AWARDED:

Master of Science in Computer and Information Science, 2001
Bachelor of Science in Mathematics and Computer Science, 1999

AREAS OF SPECIAL INTEREST:

Realistic Real-time Rendering
Models of Human Vision

PROFESSIONAL EXPERIENCE:

Research Assistant, Department of Computer and Information Science, University of Oregon, Eugene, 2000–2001

Teaching Assistant, Department of Computer and Information Science, University of Oregon, Eugene, 1999

ACKNOWLEDGEMENTS

Thank you to my advisor, Professor Gary Meyer. Without his guidance, patience and experience this thesis would not have been possible.

Thank you to my fiance Marian. Her support and insight have been priceless.

Thank you to my friend Marc Hernandez for challenging all my assumptions.

Thank you to my parents, who have always been there to support me.

Thank you to fellow students Harold Westlund and Kevin Redwine for their many interesting discussions.

This work is partially supported by grant number CCR-9988407 from the National Science Foundation to Professor Gary Meyer at the University of Oregon

DEDICATION

To my grandparents, George and Clara Pfeiffer. You've given me hope.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

Complex polygon meshes are a common occurrence in computer graphics. Sources of complex meshes include laser scanners and mesh generating algorithms like automatic surface subdivision. Applications involving meshes in computer graphics often need to be able to reduce the size of these meshes, since the performance of algorithms operating on meshes is often dependent on the number of polygons they contain. Automatic polygon mesh simplification algorithms, which convert a complex mesh into a simpler one, are in common use in the field of computer graphics. These algorithms take advantage of the fact that complex meshes often contain more information than is detectable by interaction with the mesh. While in computer graphics the method of interaction is most often visual, all but the most recent simplification algorithms are guided by geometric measures.

In their paper introducing image-driven simplification, Lindstrom and Turk present the first simplification algorithm to be guided by a mesh's visual representation [14]. To accomplish this they perform an edge collapse, render images of the mesh from many camera positions, then compare those images with images of the original mesh. The process of image comparison simulates the experience of a human observer watching each edge collapse and selecting the one that affects the appearance of the mesh the least. In this thesis, image-driven simplification is described in detail and a particular implementation of it is presented.

While the actual cost of an edge collapse in an image-driven framework is not generally limited to the region of the collapse, in this thesis we show that the error introduced by enforcing such a limit does not significantly impact mesh quality. We therefore present a pair of modifications to image-driven simplification which make the edge cost computations completely local. Local cost assignment is useful because it enables future implementations to cache local data, and it provides a smaller domain in which to pursue optimization.

Another problem commonly faced when simplifying a mesh is texture shifting caused by changes in geometry. As the mesh definition is altered, incorrect redefinition of texture coordinates can cause drastic distortion in the appearance of texture on the mesh surface. Current methods for reducing texture shifting attempt to regenerate the parameterization using a fixed texture definition. We present a technique that uses real-time solid texture to regenerate the texture definition and parameterization as simplification proceeds. A benefit of using solid texture is that the parameterization is very simple. Also, because the texture is mathematically defined, the texture shifting that does occur during simplification is predictable.

The next section summarizes previous work in the field of polygon mesh simplification. Following that, Chapter II describes Lindstrom and Turk's image-driven simplification method, and our implementation of it. Chapter III presents our extension to image-driven simplification, and compares results from the extended version with results from the original. In Chapter IV real-time solid texture is presented and an implementation is described. Chapter V summarizes the work done and presents future directions in which these techniques may be applicable.

Previous Work

Execution times for existing simplification algorithms vary from seconds to days. The most expensive algorithms are those that minimize complex energy functions. The fastest algorithms are those that can determine collapses locally with simple geometric calculations. It should be noted that a full solution to the error minimization problem for polygon meshes requires exponential time, therefore all practical methods known are approximation methods.

Previous work in mesh simplification can be divided into methods which use vertex removal, and methods which use edge collapse. Vertex removal methods simplify a mesh by first removing a vertex, then re-triangulating the hole created. Edge collapse techniques perform simplification by replacing edges with vertices, then updating the surrounding triangles to account for this replacement. The following summary is inspired by the one in [14].

Schroeder et al. use a vertex removal method in which error is assigned to vertices based on their distance to the average plane formed by adjacent vertices. In this method vertices are classified into different types based on their contribution to the mesh geometry, and each type is handled differently [19]. Ciampalini et al. maintain global error measures that represent the distance between the surfaces formed by the original and simplified meshes. They also allow edge flips during simplification, which greatly improves the quality of their results [2]. Cohen et al. maintain a pair of envelopes, one internal and one external, around the mesh. Valid removals are those whose re-triangulation does not pass through either of the envelopes [6]. Rossignac and Borrel in [18] group vertices by placing them in the cells of a grid. A representative vertex is then associated with each cell. When a collapse is considered, all vertices in the cell are replaced by the representative vertex. Their technique allows merging of previously unconnected areas of the mesh, as long as they fall inside the same cell.

The edge collapse method of Hoppe et al. works to minimize an energy function with a combination of edge collapses, edge swaps and edge splits [10]. Ronfard and Rossignac also use a method of repeated edge collapse, but they associate each vertex with a list of planes [17]. These planes are initially defined by the faces adjacent to the vertex. The cost of collapsing an edge to one of these vertices is defined as the sum of the distance of the vertex to its associated planes. Finally, the plane lists of the edge endpoints are merged together after the edge collapse occurs. Garland and Heckbert expanded upon Ronfard and Rossignac's idea by encoding implicit versions of the plane equations into a 4×4 matrix for each vertex, called a quadric [7]. Quadrics enable fast computation of the distance measure, and efficient merging of the plane equations after collapse, by summation. Garland and Heckbert also permit two points in a mesh to collapse together even if they are not connected by an edge.

Surface preserving techniques have recently become popular. Garland and Heckbert extended their quadric error metric in [8] to work with larger dimensions than the three for vertex position. Instead of using the space of vertex positions in 3D, they use vectors of vertex positions extended by vertex attributes. For n vertex attributes, a $(3+n)$-dimensional quadric is associated with each vertex. Hoppe improved upon Garland and Heckbert's surface preserving technique in [11] by introducing a more efficient method of storing the quadric error, as well as a technique for efficiently re-computing the quadric error for every collapse. This 'memoryless' simplification improves the quality of the resulting mesh, and is less memory intensive because it does not store the quadrics. Cohen et al. use a decoupled representation in which color data and texture data are stored separately from the vertex data, in color and texture maps respectively. A surface parameterization is used to map between the color and texture data and the vertices [4]. Using this representation, they are able to limit texture coordinate deviation in screen space, and provide bounds on screen space error introduced by texture shifting during simplification [5].

Most recently, Lindstrom and Turk used actual rendered images in the intermediate stages of simplification to sample the error introduced to geometry and surface properties as a result of edge collapse [14]. Image error was used to guide a memoryless collapse heuristic similar to the one used by Hoppe in [11]. Their method, called image-driven simplification, is the starting point for the work presented in this thesis. While computationally expensive, image-driven simplification can be used to guarantee results of high visual quality. It has roots in image-based rendering as well as geometric mesh simplification. This algorithm can be viewed as sampling the light field around a mesh, then consulting that light field measurement to evaluate prospective collapses. It generates simplification steps that cause little distortion of rendered images of the mesh relative to rendered images of the original. It can also allow simplifications of interior geometry and other occluded geometry that a geometric algorithm would not directly allow.

CHAPTER II

IMAGE-DRIVEN SIMPLIFICATION

Previous approaches to simplification for computer graphics have centered around achieving geometric and topological similarity. Because geometrically similar meshes tend to look similar, these methods have been able to produce pleasing results. While recent extensions to geometric methods have allowed simplification to take parameterized surface properties into account, such as vertex colors and texture coordinates, these methods still only indirectly measure changes in appearance.

The goal of image-driven simplification is to maximize the visual similarity between a base mesh and a simplified one by directly sampling image difference. Visually similar objects are objects that appear identical to each other when viewed by a human observer. While the type of image difference metric affects how accurate this method is, even the simplest type of image difference metric can lead to simplified meshes of high quality. Image-driven simplification as implemented by Lindstrom and Turk is sensitive to surface detail, and can measure changes due to color, texture, geometric detail, and silhouette affects.

A secondary aim of image-driven simplification is efficient execution. The method is inherently expensive due to repeated comparisons of many images to evaluate error, so several optimizations have been implemented by its authors. These include restricting the update of error in an image to regions in which error is introduced, use of a limited number of sampling points, and lazy evaluation of edge error effects caused by edge collapse.

Overview

Lindstrom and Turk's method of image-driven simplification is similar to existing edge collapse simplification methods in that it uses a greedy algorithm to select edges. First the algorithm computes and stores edge collapse costs for all the edges. Then it repeatedly extracts the minimum-cost edge collapse, performs it, and updates the affected edge collapse costs. The algorithm terminates when the target number of faces is reached.

The image-driven approach differs from previous methods in how it evaluates the cost of edge collapses. With geometric methods, edge collapse cost is a function of a mesh's geometry. In image-driven simplification, edge collapse cost is a function of multiple images of the rendered mesh before and after collapse of an edge. Section 2.2 more closely examines how this is achieved.

Running times for the image-driven technique lengthen as the number of images used to sample the mesh increase. Running times are also dependent on the time required to render the images. This effectively limits the technique to meshes and rendering methods that can be considered in real-time. Lindstrom and Turk use hardware-accelerated *OpenGL* rendering with flat or smooth shading, texture mapping, and a single light source.

Sampling the Mesh

To sample the appearance of a mesh, an arrangement of virtual cameras is selected so that every polygon is visible from at least one camera. One technique to do this is to arrange the cameras on a sphere centered at the mesh's center, with the complete mesh in the view frustum of each camera. This provides an even sampling of the mesh from

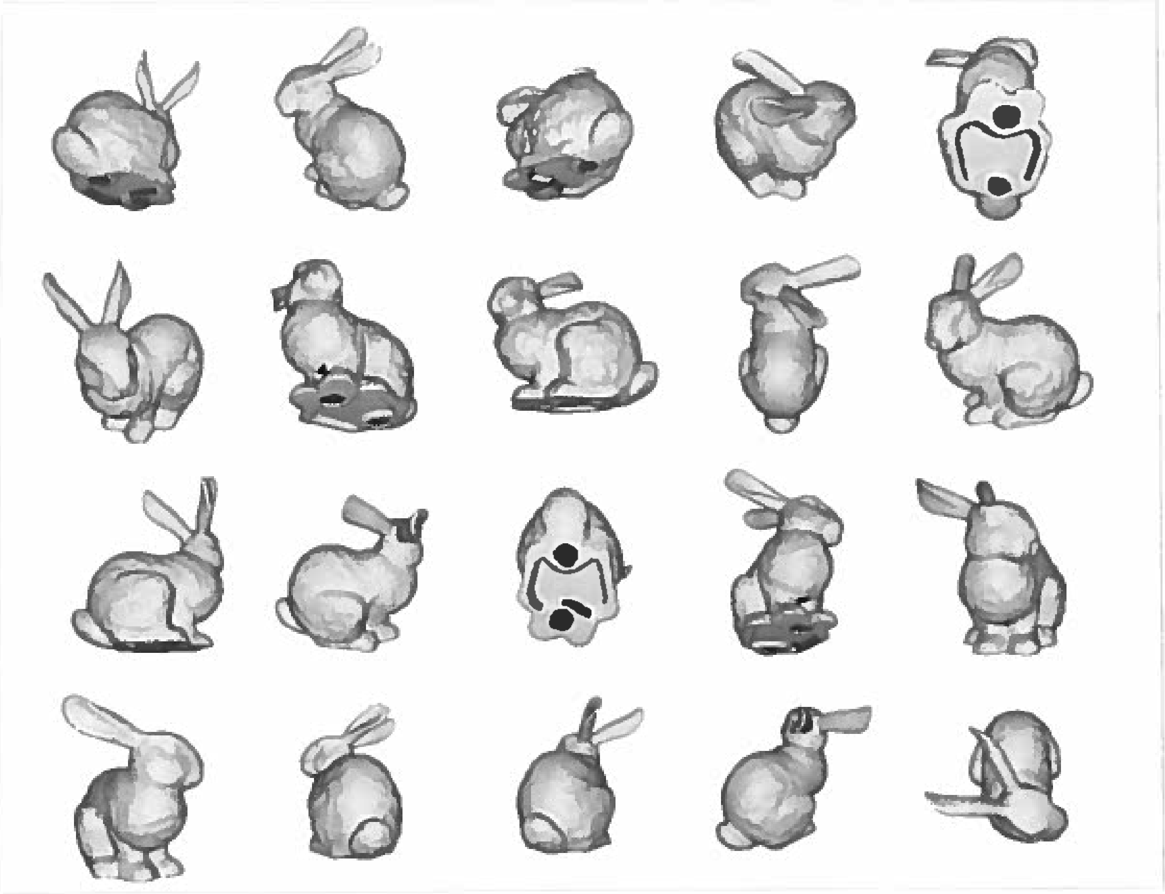


FIGURE 1. The 20 sampling views of the Stanford bunny.

each direction. Enough cameras must be used to prevent aliasing of the collected error values. Aliasing is prevented by averaging the error over several cameras, so in fact each polygon should be visible from several cameras. Lindstrom and Turk used 20 cameras arranged on a sphere to simplify the Stanford bunny mesh. Following the idea of a figure in [14], Figure 1 shows the Stanford bunny from these 20 viewpoints.

An appropriate resolution must be chosen for the sample images. While it is not generally feasible to use as high a sampling resolution as a final viewing resolution, using a lower resolution may introduce error. If the area affected by a collapse is less than a pixel in the sampling resolution, but not in the final viewing resolution, the algorithm may allow a collapse that is of relatively high error when subsequently viewed. Lindstrom and Turk use a sampling image size of 256 × 256 and a final viewing size of 512 × 512.

Once a set of images $\mathcal{Y}^0$ is collected for the original mesh and a set $\mathcal{Y}^k$ is collected after the k^{th} prospective edge collapse, an image based error can be associated with an edge. First, however, an image difference metric must be selected. Lindstrom and Turk used mean squared error which, while not a metric in a strict sense, provides an ordering on image difference. In general any image difference metric could be used. For a mesh after k edges have been collapsed, the mean squared error associated with an edge is:

$$d_{MS}(\mathcal{Y}^0, \mathcal{Y}^k) = \frac{1}{lmn} \sum_{h=1}^{l} \sum_{i=1}^{m} \sum_{j=1}^{n} (y^0_{hij} - y^k_{hij})^2. \tag{2.1}$$

Optimizations

Image comparison is the most expensive step of image-driven simplification, so Lindstrom and Turk attempt to limit the number and cost of image comparisons done. First, they only difference images in the region of error collapse. This prevents computing most differences that would end up being zero. Next, instead of updating the error introduced to surrounding edges immediately, they flag these edges. Subsequently, only when a flagged edge is found to be the cheapest is its error updated. Because edge collapses tend to increase the error associated with neighboring edges, this kind of lazy evaluation will introduce only a small amount of distortion to the mesh. Finally, Lindstrom and Turk rely upon hardware support for the *OpenGL pixel buffer* extension to enable fast access to the frame buffer.

Since affected edges are mostly near the collapsed edge, adjacency information is calculated many times during a collapse. A mesh representation that stores edge information is required for good performance. Unlike most edge collapse methods, edges far away from a collapsed edge may have their error affected. However, Lindstrom and Turk showed that by ignoring distant edge error their technique still chose edges that were in the best 0.1% of edges on average.

Limitations

Using rendered images as a sampling medium may restrict the applicability of the resulting mesh. Since geometric fairness is not guaranteed under image-driven simplification, meshes generated by it may not be appropriate for collision detection or other geometry-based calculations. The resulting images can vary between rendering hardware, causing the simplified mesh to look different from the original on different workstations. Finally, because a simplification is done with a specific lighting condition the resulting mesh may not look similar to the original under different lighting conditions.

Because every edge must be collapsed and an image evaluated to determine error, the running time of image-driven simplification can be prohibitive for large meshes. Running time only varies linearly in the product of the number of faces in the mesh and the number of cameras and the resolution used. However, since the constants for image comparison are large, each edge collapse is expensive. For very large meshes, many triangles will project to the size of a pixel, so the image-driven approach actually wastes a good deal of effort on these meshes by comparing images that are equal. Lindstrom and Turk instead pre-simplify the mesh with a fast geometrically-based algorithm, then refine that mesh with image-driven simplification to get a high quality simplified version.

An Implementation

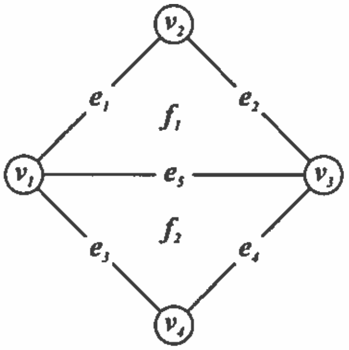
Goals

This section describes our implementation of Lindstrom and Turk's image-driven simplification method. The primary use of this program is as a test platform for extensions to image-driven simplification, so some details of our implementation differ from that of Lindstrom and Turk. Most notably, our implementation treats key elements of image-driven simplification abstractly, including the edge collapse methodology, the image metric, and the sampling-camera placement. Furthermore this implementation works on consumer level hardware, requiring only hardware-accelerated video card supporting *OpenGL*.

Some of the optimizations used by Lindstrom and Turk are omitted from our implementation in the interest of maintaining an abstract system. Optimizations not included are partial image updates (and therefore usage of the *pixel buffer* hardware extension to *OpenGL*) and lazy evaluation of edge costs. However, to enable complete simplifications and to successfully test the method some performance issues are addressed. Fast queries of an edge's neighboring vertices are available via an edge connectivity data structure. Repeated edge cost evaluations are avoided by storing edge costs in a table, and updating this table with the local error introduced to edges near an edge collapse.

Implementation

Our implementation of image-driven simplification is described here in a bottom up fashion. First, the subroutines that do the work are given. These subroutines embody



=

Vertices:
$v_1 = e_1\ e_5\ e_3$
$f_1\ f_2$
$v_2 = e_1\ e_2$
f_1
$v_3 = e_2\ e_5\ e_4$
$f_1\ f_2$
$v_4 = e_3\ e_4$
f_2

Edges:
$e_1 = v_1\ v_2$
$e_2 = v_2\ v_3$
$e_3 = v_1\ v_4$
$e_4 = v_4\ v_3$
$e_5 = v_1\ v_3$

Faces:
$f_1 = v_1\ v_2\ v_3$
$f_2 = v_3\ v_4\ v_1$

FIGURE 2. The mesh edge data structure, with vertex list, edge list, and face list.

the edge connectivity data structure, the method of collapse, details of camera placement, the method of edge cost evaluation, and a technique for prevention of repeated computation. Lastly, the routine that uses the above subroutines to achieve image-driven simplification is given. Along the way the differences between our implementation and that of Lindstrom and Turk are discussed.

Mesh data is stored in a structure that maintains connectivity information for edges. A winged-edge and half-edge implementation were considered, but those would limit meshes to being manifold surfaces. Instead, a custom structure was written that supports non-manifold surfaces. This structure contains a list of vertex objects, a list of face objects, and a list of edge objects. Each vertex object contains a list of the indices of edges incident to it, and a list of indices of faces incident to it. Face objects have a list of indices of the vertices that make up the face. Edge objects have a list similar to that for faces. With this structure, connectivity information can be retrieved in constant time. See Figure 2 for an example of this data structure.

The above mesh structure supports edge collapse, and subsequent re-expansion of that collapse. When doing an edge collapse, an edge is collapsed to a vertex, and that

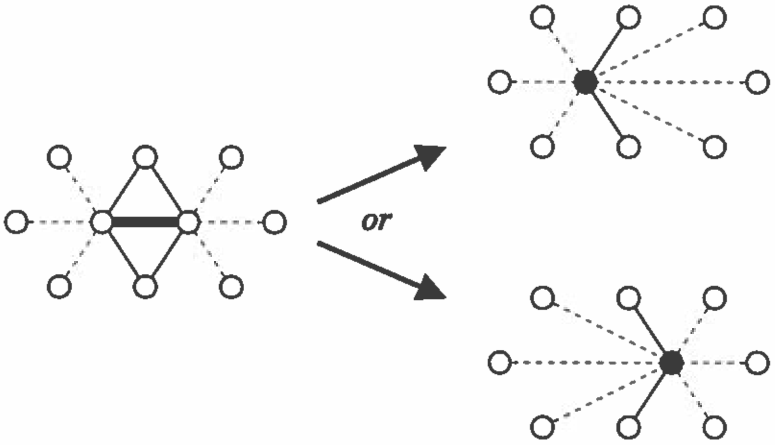


FIGURE 3. The new vertex is placed at random at either of the two original edge vertices.

new vertex must be placed somewhere. While Lindstrom and Turk use a volume preserving calculation to place the new vertex, our technique randomly chooses one of the existing vertices as the new vertex position, as shown in Figure 3. The memoryless approach tends to produce better meshes, but the random choice approach has been shown to have good performance as well. Contrary to intuition, using the midpoint between the vertices of the collapsed edge is actually the worst simple placement choice. Lindstrom and Turk provide an excellent comparative analysis of these approaches in [13]. Using existing vertices may also be beneficial for hardware accelerated rendering using progressive meshes that are generated with this method, since a single immutable vertex list can be used.

Instead of an automatic camera placement routine, here the camera placement is done by the user specifying each camera explicitly. This allows more flexibility to experiment with camera positioning. A camera placement was determined for the Stanford bunny that matches the one used by Lindstrom and Turk. For n cameras, the set of images $\mathcal{Y}^i = \{y_{i0}, \ldots, y_{in}\}$ is generated by rendering the mesh for each camera view and

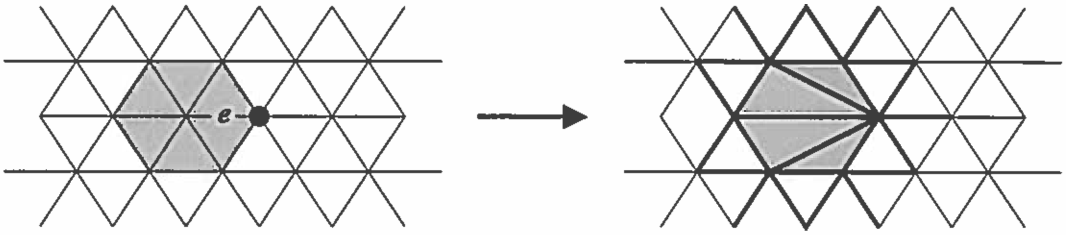


FIGURE 4. Shaded triangles are directly affected by collapse of edge *e* to the darkened vertex. After collapse, the error associated with the darkened edges on the right must be updated.

copying the frame buffer of *OpenGL* to one of n arrays in memory.

To avoid repeated computation of edge costs, these costs are stored in a table. After an edge collapse, the costs of edges affected by that edge collapse are updated. Since we have made the simplifying assumption that the effect of an edge collapse on edge cost is limited to a local region around the collapsed edge, significant performance gains can be realized by a table update. The exact set of edges that gets updated on collapse of an edge e is all edges that contain at least one vertex that is part of a triangle touching a vertex of e. This set is illustrated in Figure 4. To retrieve the minimum cost edge, a linear search is performed on the table to find the minimum cost edge to collapse.

Now that the individual subroutines are in place, they can be combined with the greedy strategy outlined in Section 2.1. Pseudocode for our image-driven collapse routine is as follows:

```
COLLAPSE-EDGE(target, mesh)
edgeErrorTable ← NIL
minErrorEdge ← $\infty$
affectedEdgeList ← NIL
while NUM-FACES(mesh) < target
do if FIRST-COLLAPSE()
      then edgeErrorTable ← TEST-COLLAPSE-ALL(mesh)
   minErrorEdge ← FIND-MIN(edgeErrorTable)
   affectedEdgeList ← COLLAPSE-EDGE(minErrorEdge, mesh)
   UPDATE-ERROR(affectedEdgeList, mesh)
   DELETE(minErrorEdge, edgeErrorTable)
```

This algorithm is greedy because at every stage it selects the lowest-cost edge to collapse. When the loop exits, the mesh contains the simplified mesh.

Results

Simplifications reported in this thesis were performed on a desktop computer with a 900MHz AMD Athlon processor, 256MB of RAM and an NVIDIA GeForce2 GTS video card with 32 MB of RAM. Timings reported by Lindstrom and Turk in [14] were done on a 250 MHz R10000 Silicon Graphics Octane workstation with 256 MB of ram and IMPACTSR graphics. We used a sampling resolution of 128×128 for images.

Figure 5c shows the output of our implementation of image-driven simplification using the Stanford bunny. The bunny was simplified from 65,451 faces to 1,336

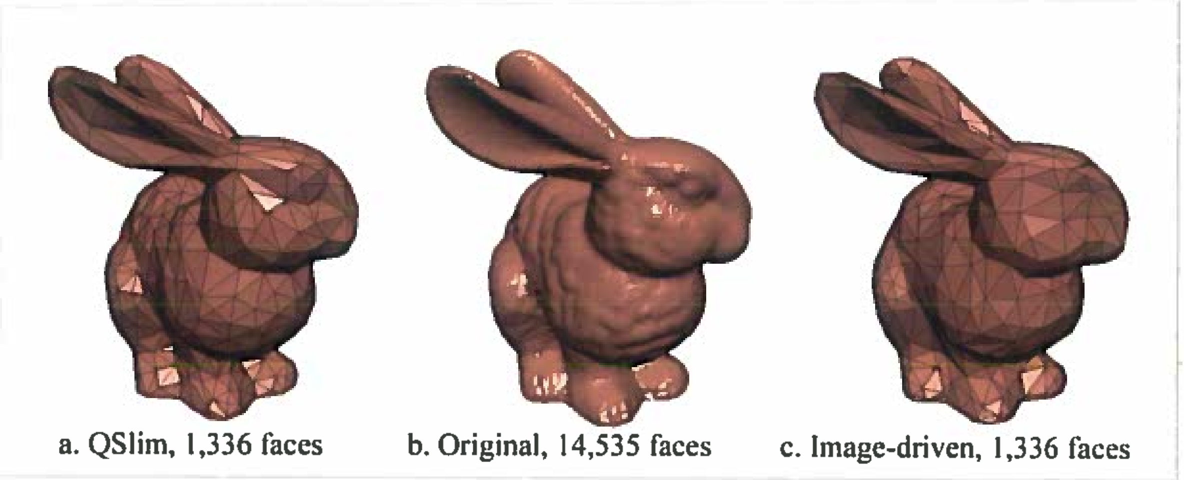

a. QSlim, 1,336 faces b. Original, 14,535 faces c. Image-driven, 1,336 faces

FIGURE 5. The center mesh is the base mesh. Note the smooth edges on the bunny's ear in our result in c, along with the drastic simplification of the bunny's face.

faces, by first using QSlim[1] to simplify from 69,451 to 14,535 faces, then using our implementation of image-driven simplification to simplify from 14,535 to 1,336 faces. For comparison, Figure 5a shows the result of QSlim simplifying from 69,451 to 1,336 faces, while Figure 5b shows the mesh with 14,535 faces.

In Figure 6 we compare the results of our implementation of image-driven simplification against the implementation presented by Lindstrom and Turk in [14]. Figure 6a is an image taken from their image driven simplification result for 1,336 faces. Again, Figure 6b is again the mesh with 14,535 faces and Figure 6c is our result with 1,336 faces. Figure 6a took Lindstrom and Turk's method 970 seconds to compute. We can see that our implementation made some simplifications similar to Lindstrom and Turk's method. In particular, the silhouette is well-represented, at the expense of interior detail.

From inspecting Figure 5 we can see that image-driven simplification preserves the silhouette better than QSlim, the geometric technique. Also, the haunches and ears of the bunny appear to be more accurately preserved by this implementation of image-

---

[1] QSlim v2.0 was used, with options `-B 10`.

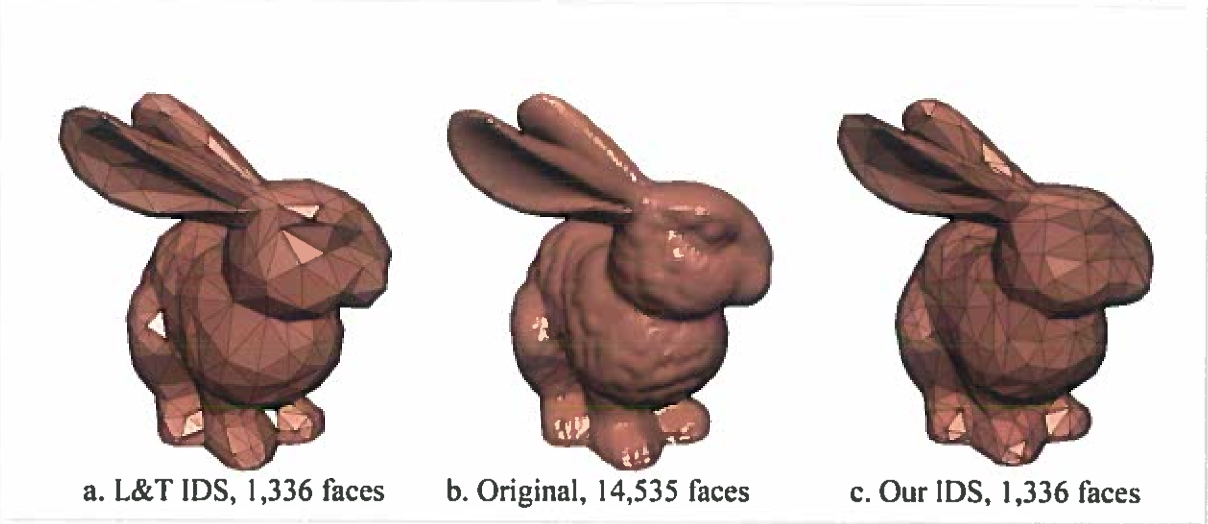


FIGURE 6. Left: Lindstrom and Turk's result reprinted from [14]. Right: our result.

driven simplification than by QSlim. The algorithm allowed simplification of surface detail, but kept the silhouette relatively well defined. These tendencies are similar to those reported by Lindstrom and Turk in [14].

CHAPTER III

SHARD-DRIVEN SIMPLIFICATION

Goals and Introduction

Lindstrom and Turk's image-driven simplification method restricts computation of image difference to a region around the collapsing edge, but for every prospective edge collapse renders the entire mesh from every camera position. There are a couple of reasons why rendering the whole mesh is undesirable. If an expensive shading model is being used, the cost of evaluating it on all the triangles in the mesh could be unacceptably high. Also, many cameras are required to sample complex meshes. This is particularly important for self-occluding meshes, since any collapse must be visible from several cameras. In this chapter we describe an implementation of a local edge cost evaluation technique that addresses these issues.

It is possible to specify the set of all triangles that can contribute to the cost of collapsing a particular edge. A triangle contributes to the collapse cost of an edge e if the appearance of the rendered version before collapse of e differs from the appearance of the rendered version afterward. In practice there are a limited number of ways that such a contribution can be made. One possibility is that the geometry of the triangle can change. In this case the pixels that make up the triangle in the rendered image may either be shaded differently or occur in a different place from those of the original. Another possibility is that the pixels of a triangle may have been visible before a collapse, but made invisible after the collapse, or vice-versa. Only triangles which experience a change in geometry or visibility can contribute to the collapse cost for an edge.

We propose that only the set of triangles that can contribute to the cost of collapse of an edge e need to be rendered to compute the cost of e in an image-driven scheme. We call this set of triangles a *shard* for e, and hereafter refer to the cost of collapse of e as simply the *cost* of e. In the following section, shards are more restrictively defined.

By limiting the rendering during edge cost calculation to a shard, the cost of evaluating an expensive shading model is minimized. If shards are localized and consist of a small number of triangles, self-occlusion is rare and it is easier to guarantee every face is seen by a camera. Finally, a shard-based approach may allow the consideration of edge costs for shards independent of a particular instance of a mesh.

Implementation

Our implementation computes the cost of collapse of an edge as its effect on a shard of triangles whose geometry is altered by that collapse. The method of greedy edge choice is the same as in image driven simplification and is reused. The mesh data structure is retained but expanded upon, and all the edge placement and collapse routines are unchanged. Additions include new edge cost evaluation routines and an updated mesh data structure.

This section describes the updated mesh data structure and the new edge cost evaluation routines. Following Section 2.5, we describe the technique from the bottom up. A technique to generate shards from mesh data is given, then routines to compare shards to each other are specified. We show how these subroutines are combined together to evaluate edge cost, and how the new edge cost result is used with the greedy strategy of Chapter II to perform shard-driven simplification.

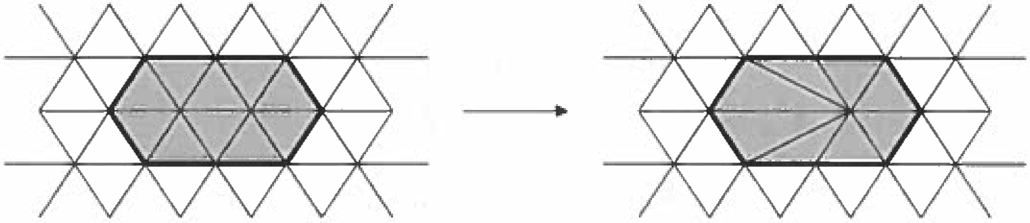


FIGURE 7. The dark line shows the perimiter of the shard. It is unaltered by collapse of the shard's associated edge

Shard Generation

Shards in this implementation are more restricted than those defined above. Here a shard is composed only of triangles that directly experience geometric changes as a result of the collapse. For an edge e between vertices v_1 and v_2, this set is composed of the triangles that share a vertex with v_1 or v_2, as in Figure 4. No triangles outside this set are geometrically affected by the collapse of e. The perimeter of this set of triangles is fixed under collapse of e, as shown in Figure 7.

It is important to note that triangles outside of the aforementioned set may contribute to the actual cost of collapsing e. These triangles are exactly those that experience a change in visibility as a result of the collapse of e, but whose geometry is not changed. By rendering only shards, the contribution of these triangles is replaced by a contribution from the background color as in Figure 8. In this way using shards introduces error into the edge cost calculation

The error introduced can be an over-estimation or under-estimation of what the difference would be if the full mesh was used. An over-estimation occurs if the far triangles are closer in intensity to the near triangles than is the background intensity. An under- estimation occurs if the intensity of the background and near triangles are more similar than those of the near and far triangles. In Figure 9 we see an over-estimation

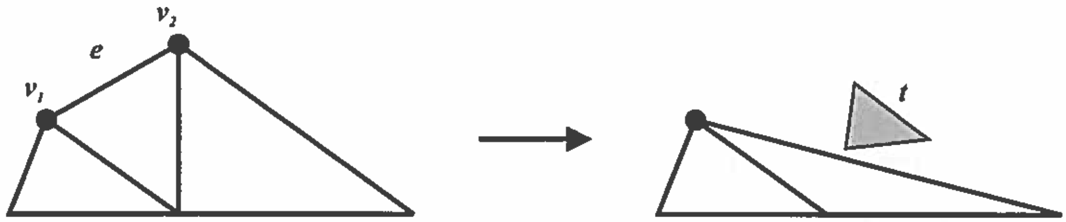


FIGURE 8. Collapse of e to v_1 should reveal triangle t. However, the contribution of t (the shaded region) will be replaced by the background color when just the shard is rendered.

and in Figure 10 an under-estimation. Using a directional light source aligned with the camera, far and near triangles will be shaded similarly along a line of sight, thus over-estimation of edge cost is more likely. Positional light sources that include an attenuation factor will cause some over-estimation and some under-estimation. Our implementation uses directional light sources, without attenuation, to minimize the effect of this type of error.

We modify the mesh data structure described in Chapter II to generate a shard for an edge e by consulting the lists of triangles stored with each vertex of e. These lists are merged into a list of triangles that form the shard. The shard itself is a mesh, thus edge collapses can be performed on it and it can be rendered.

Shard Comparison

Shards in shard-driven simplification are compared in much the same way as the full mesh is compared in image-driven simplification. Multiple cameras are used, in the same positions as in traditional image-driven simplification. There is a difference, however, in how the images themselves are compared. In image-driven simplification images of the original mesh are compared with images taken after each prospective collapse. But finding the set of triangles in the original mesh that corresponds to a shard

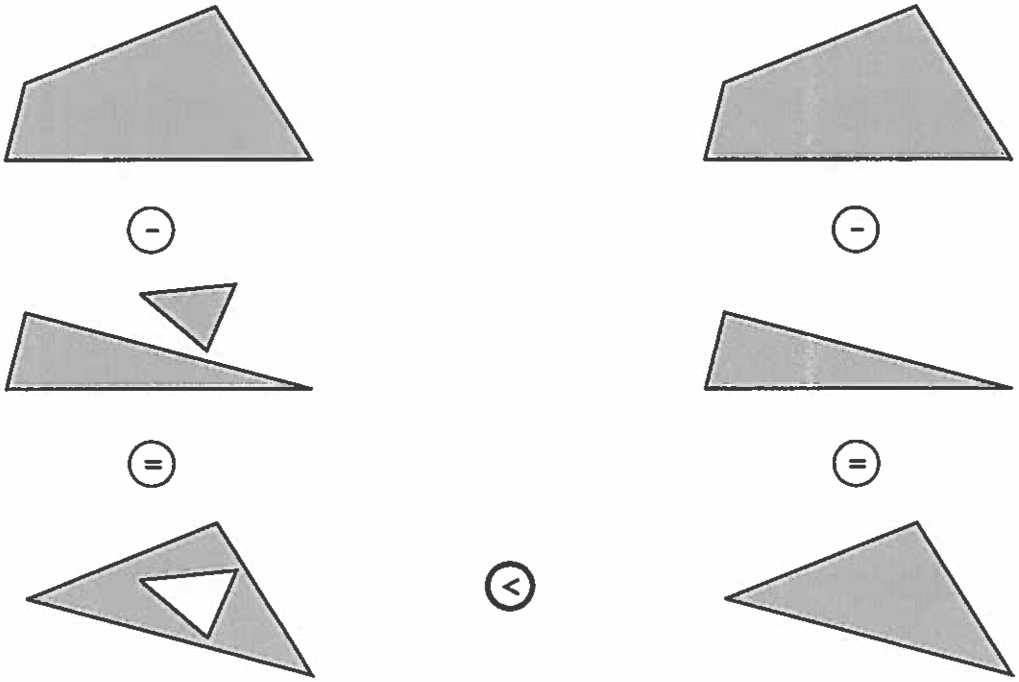


FIGURE 9. At left, comparison using image-driven simplification. At right, using shard-driven simplification. When the intensity of the background is similar to that of the occluding geometry, the error is over-estimated.

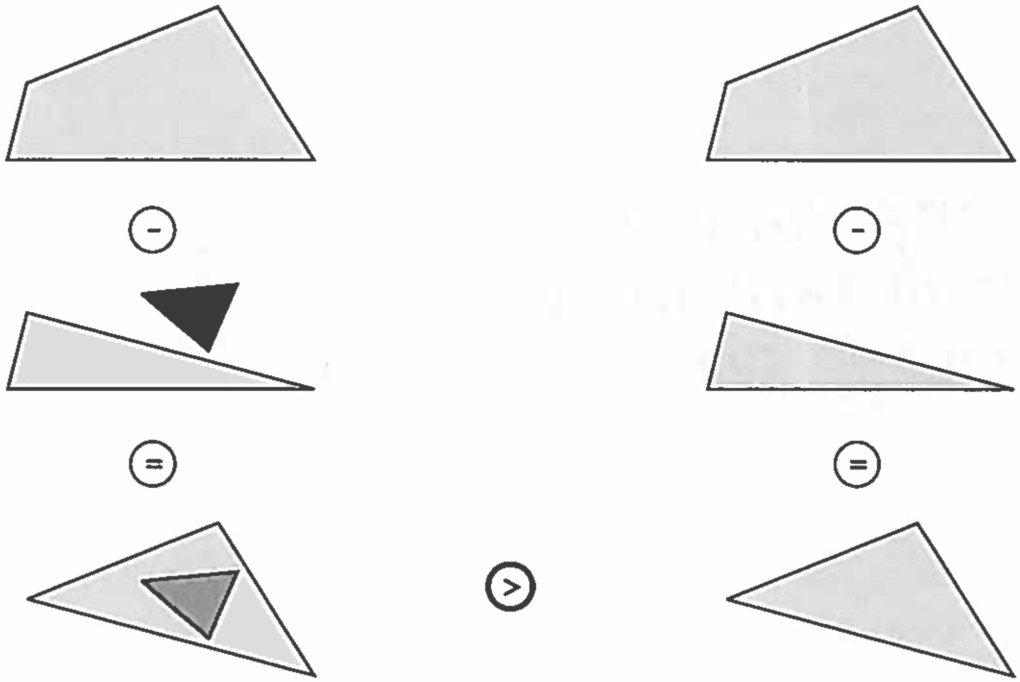


FIGURE 10. At left, comparison using image-driven simplification. At right, using shard-driven simplification. When the intensity of the occluded triangle is similar to the background, the error is under-estimated.

in a partially simplified version may be difficult. Furthermore, the images of shards from the original mesh would have to re-generated for every prospective edge collapse, so there would be no performance benefit gained by comparing against the original mesh.

To avoid the complications described above, we deviate from the technique used by Lindstrom and Turk and instead evaluate image differences incrementally. Images of the mesh are saved just before a collapse, then compared with images of the mesh after a collapse. Figure 11 compares the two techniques. In incremental evaluation a set of images $\mathcal{Y}^{k-1}$ is generated before the k^{th} edge collapse. To evaluate the cost of collapse of an edge e, e is collapsed and a set of images $\mathcal{Y}^{k}$ is generated. The cost of collapsing e is $d_{MS}(\mathcal{Y}^{k-1}, \mathcal{Y}^{k})$, with d_{MS} the same as in Equation 2.1. The failure to maintain images of the original mesh to compare against may contribute to a loss of fidelity in this implementation.

Edge Cost Evaluation

Evaluation of edge costs is performed differently than in image-driven simplification. Instead of computing prospective collapses for each edge in the overall mesh, we extract the shard associated with each edge, and collapse the edge in the shard that is equivalent to the edge in the original mesh. This eliminates the step of expanding the collapsed vertex into an edge after a prospective edge collapse.

Given the shard collapse and shard comparison subroutines, shards are now used to evaluate the cost of an edge e as a candidate for the k^{th} edge collapse. This process is illustrated in Figure 12. First the shard P associated with e is extracted from the mesh, where P has an edge e' that is equivalent to e in the original mesh. Shard P is rendered

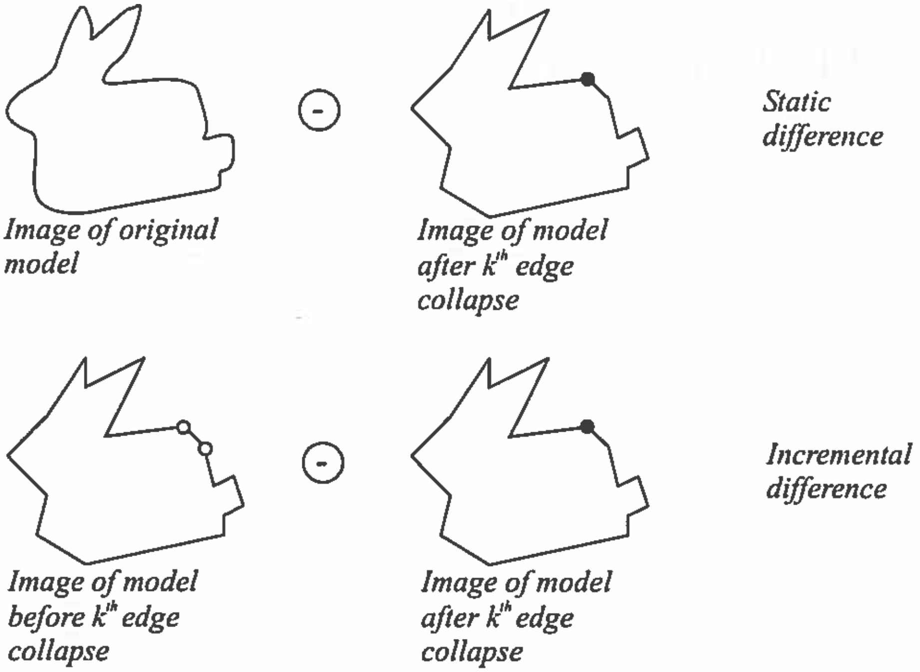


FIGURE 11. In static difference, the image of the original mesh is maintained. In incremental difference, only the image of the mesh before and after collapse are used.

from each of the camera viewpoints, with the resulting images stored as the set $\mathcal{Y}^{k-1}$. Next e' is collapsed in P to form a mesh P', and the set $\mathcal{Y}^k$ is generated by rendering P' from all the camera viewpoints. Finally, the cost of e is computed using the incremental difference routine on P and P'.

Integrating With The Greedy Solution

To use the greedy strategy given in Chapter II, we replace the use of prospective edge collapses in the pseudocode in Section 2.5 with the shard based edge cost evaluation described above. Edge collapse costs are still stored in a table, and the same randomized endpoint edge placement algorithm for the edge collapses is employed. Using this randomized approach requires remembering which endpoint was randomly selected in the test collapse and using that one for the actual collapse.

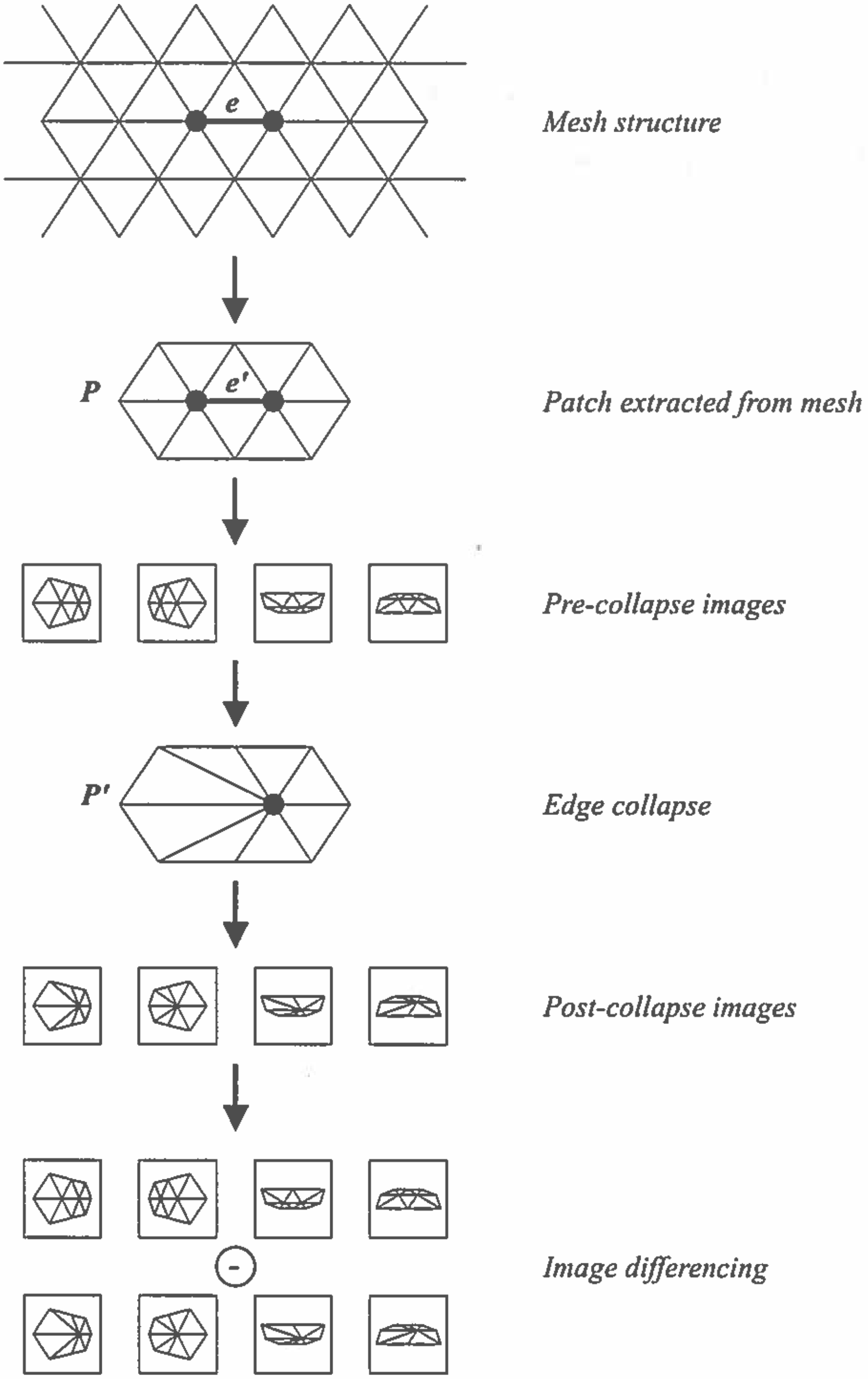


FIGURE 12. The sequence of operations for incremental difference with shard-based simplification.

A Side Effect: Simplification of Interiors

Lindstrom and Turk went to considerable effort to extend image-driven simplification to prevent simplification of hidden interiors in [14]. With shard-driven simplification, such simplification is prevented automatically. When triangles are rendered a shard at a time, there is little chance for occlusion to occur. This has the effect of preventing simplification of hidden interiors, since interior geometry is always visible for an interior prospective edge collapse. The removal of self-occlusion effects makes defining sampling cameras easier, because it is no longer necessary to position cameras to sample around occluding geometry.

An extension to shard-driven simplification is possible that would allow simplification of hidden interiors. This would entail rendering all faces of the mesh except those in the shard with the background color. Such an effect could allow faster simplification than traditional image-driven simplification if a complex shading model was being used, since most faces of the mesh could be rendered as the background color using a simple shading model instead.

Results

The following are conclusions about the shard-based approach, and results of simplification using it, compared to other methods. The images in this section were generated on the same system as specified in Section 2.5.

Running Time

In Table 1 the running times for QSlim, Lindstrom and Turk's implementation of image-driven simplification, our implementation of image-driven simplification and

TABLE 1. Compared running times for QSlim, Lindstrom and Turk's of Image-driven Simplification, and our Image-driven and Shard-driven simplification.

Method	QSlim	L and T image-driven	Our image-driven	Our shard-driven
Time	4 seconds	970 seconds	34,577 seconds	6,611 seconds

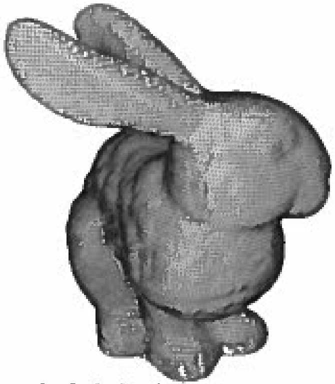
shard-driven simplification are given. Both of our methods are largely unoptimized. Again, the goal of this work is to compare mesh quality, under the assumption that optimizations similar to those Lindstrom and Turk used will be applicable to our technique in the future.

Shard-driven simplification took 6,611 seconds to simplify the Stanford bunny mesh in Table 1, while our implementation of image-driven simplification took 34,577 seconds, as in Chapter II. This occurred because the flat shading model used here is very fast compared to the image comparison step. Hence the time gained by restricted rendering is enough to overcome the extra time used in extracting shards from the base mesh to evaluate collapse costs. We believe that using a technique like Lindstrom and Turk used in [14] to restrict the area of image difference computation would make the shard-driven method even faster.

Mesh Fidelity Compared

Figure 13 shows a wireframe version of the Stanford bunny mesh simplified from 14,535 to 1,336 faces under QSlim, and our implementations of image-driven simplification and shard driven simplification. Our implementation of image-driven simplification dedicated more triangles to silhouette edges than did QSlim, particularly around the ears and base of the mesh. However, for silhouette preservation the shard-driven technique is clearly superior, having dedicated far more triangles to the silhouette edge than ei-

a. QSlim, 1,336 faces b. Image-driven, 1,336 faces c. Shard-driven, 1,336 faces

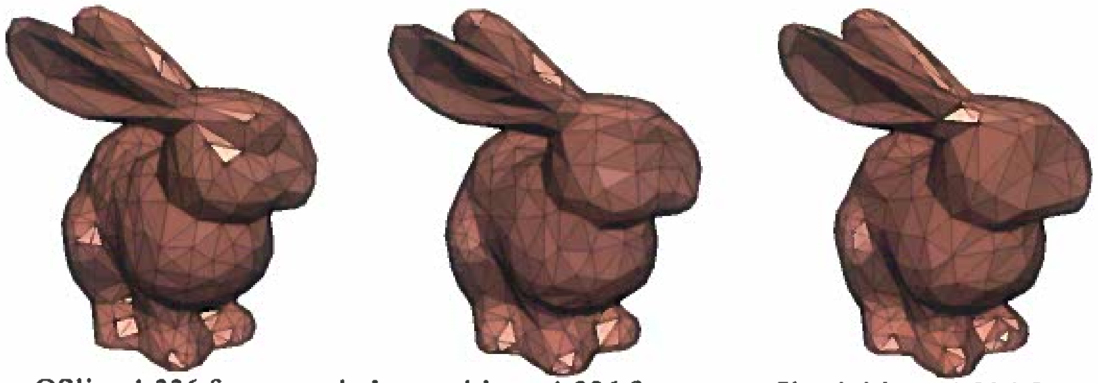


d. Original, 14,535 faces

FIGURE 13. Wireframe rendering of the output of QSlim, our image-driven simplification, our shard-driven simplification, and the base mesh.

ther QSlim or the original image-driven simplification technique. Note the particularly well-rounded ears, feet and tail.

In Figure 14 the distribution of triangles in non-silhouette surface geometry is more apparent. After shard-driven simplification relatively few triangles are used to represent the flat portions of the Stanford bunny. QSlim and image-driven simplification appear to have more faithfully captured the variations in geometric detail on the non-silhouette surfaces of the bunny such as the eyes and haunches. The shard-driven technique spent more resources on the silhouette, at the expense of the non-silhouette geometry.



a. QSlim, 1,336 faces b. Image-driven, 1,336 faces c. Shard-driven, 1,336 faces

d. Original, 14,535 faces

FIGURE 14. Flat shaded rendering of the output of QSlim, our image-driven simplification, our shard-driven simplification, and the base mesh.

a. QSlim, 1,336 faces b. Image-driven, 1,336 faces c. Shard-driven, 1,336 faces

d. Original, 14,535 faces

FIGURE 15. Another viewpoint of the result from QSlim, our image-driven simplification, and our shard-driven simplification.

Figure 15 shows how well the silhouette was preserved for the shard-driven method from a different viewpoint. While in Figure 14 the shading of the haunches is less accurate after shard-driven simplification than QSlim or image-driven simplification, the silhouette of the head of the bunny was better preserved by the shard-driven technique. Also the curvature of the inside part of the ear appears more well defined after shard-driven simplification than after the other techniques.

The wireframe and shaded renderings of the Stanford bunny created with the shard-based technique appear to be of high quality when compared to those created by Image-driven Simplification and QSlim. The shard-based technique spent more resources to preserve the mesh silhouette, and less to preserve interior detail. This effect is even more pronounced than in our implementation of image-driven simplification. It is clear that both image-driven simplification and shard-driven simplification produce meshes of comparable or higher quality than QSlim.

Summary

Using the a shard-driven approach to image-driven simplification appears to perform no worse than image-driven simplification, and can produce meshes with better defined silhouettes, at the expense of representation of non-silhouette surface details. This feature is in the spirit of image-driven simplification, as Lindstrom and Turk in [14] cited enhanced silhouettes and simplified surface detail as a beneficial feature of their approach. Furthermore, this quality was achieved using incremental comparison which is a lower quality method of image comparison than that used by Lindstrom and Turk.

Using shards instead of the full mesh allows the experimenter more freedom to experiment with effects using image-driven approaches to simplification. We also feel it makes image-driven simplification more like geometric edge collapse methods, since it forces the calculations to be local. One could implement an out-of-core image-driven simplification technique using shard-driven simplification with only a few modifications, since rendering does not require the full mesh. See [12] for a description of current methods for out-of-core polygon mesh simplification.

Future Directions

The local nature of shards should also allow a future implementation to consider shards that are transformed away from their position on the original mesh, and considered in isolation. Then, a single array of cameras and a single lighting condition could be used in place of the hemispherical camera and light set of image-driven simplification. Individual shards would be successively moved in front of these cameras and transformed to face them in a predictable way, then evaluated.

Given such a system, we envision a process of storing the error associated with certain representative shards in a database. When a similar shard is encountered elsewhere in the mesh, its error value can be retrieved from the database and used in place of an explicit image differencing operation. This technique has the potential of greatly accelerating the use of complex and expensive image metrics in image-driven simplification, particularly in regularly sampled meshes with many triangles. Such meshes are commonly generated by three dimensional scanning equipment.

This technique poses several challenging implementation problems. Driving most of these problems is the need for a database of spatial information that has small access times. Each shard, which may in some cases be quite irregular, must be transformed to face the bank of sampling cameras. Furthermore, a sampled shard must be stored with sufficient auxiliary information to allow it to be found by the lookup function of the database. A statistical evaluation of the geometric content of some representative meshes would be useful to predict what kind of compression would be possible in this database as well.

We note that this technique has an analog in fractal image compression, where a subset of small pieces of an image are rotated, translated, and scaled to reproduce that image. Fractal image compression is described in [3]. In our technique, a subset of the shards that form a mesh would be rotated, translated and scaled so that their combined error would form the total error of the mesh as the result of any edge collapse.

CHAPTER IV

REAL-TIME SOLID TEXTURE

Motivation

This chapter discusses a technique that uses *OpenGL* hardware accelerated texture mapping to render meshes with solid texture in real-time. Meshes that use texture in *OpenGL* have vertices with texture coordinates as well as position. When an edge collapse happens during mesh simplification, nearby texture coordinates must be updated along with vertex positions. This task is made more difficult by the fact that a simplification is an action in three dimensions, while texture coordinates are two dimensional. Inaccuracies of texture coordinate assignment are manifested as shifts of texture content. Image metrics and mean squared error calculations are very sensitive to subtle shifts in image content, hence they tend to overestimate the error such shifts represent. To improve the quality of simplified meshes that use texture, a good texture placement scheme is needed.

One way to simplify the problem is to move texture coordinates into three dimensions. Here, a good placement routine for vertex positions will translate directly into a good placement routine for texture coordinates. Three dimensional texture coordinates lead naturally to the notion of solid texture, as introduced by Peachey in [15] and Perlin in [16]. However, these techniques as given are too slow and memory intensive to use in a real-time system. For real-time solid texture we turn to the method introduced by Carr et al. in [1], which achieves real-time rendering of solid texture by packing the solid texture data into mesh-specific two dimensional texture maps.

We have implemented a simple version of the real-time solid texturing method of Carr et al. This implementation provides insight into future texture parameterization solutions for mesh simplification, and generates appealing images in real-time.

Implementation

Overview

Carr et al. store the solid texture contribution in a two dimensional texture map tailored to the mesh. Since the meshes considered are composed exclusively of triangles, it suffices to sample the contribution of the solid texture to a number of points on each triangle. These contributions are then stored together in a single two dimensional texture map. By using traditional textured rendering with this texture map, current rendering hardware can render solid textured meshes in real-time.

There are a number of technical issues to be resolved in order to get good quality solid texturing with this method. The first is efficient use of texture memory. Memory is a limited resource, and each triangle must have its own dedicated piece of texture memory in the texture map. This places restrictions on the sampling density for triangles. Also, irregularly-sized and stretched triangles complicate texture packing, and can introduce distortion artifacts in the rendered mesh.

Our implementation of real-time solid texturing uses the simplest sampling method, and hence the simplest texture packing scheme. We take the same number of samples for each triangle, producing identically-shaped and sized texture data elements for them. These data elements can then be easily packed together into a single texture since they all have the same size and shape. We use hardware accelerated *OpenGL* for the final rendering.

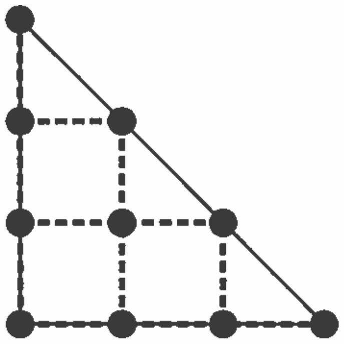


FIGURE 16. Sampling grid for S=4. F is evaluated at the 10 grid crossing points, shown darkened here.

Sampling

Solid texture sampling is an offline process that produces a texture map for a given mesh. This texture map stores samples of the surface of the mesh taken from within the solid texture object. Since the meshes we consider are all composed of triangles, we only describe how our implementation samples triangles in the solid texture space. Meshes with more complex primitives than triangles can be tessellated to consist only of triangles, so our algorithm is not particularly restrictive in the meshes it can be applied to.

A solid texture in our implementation is defined as a function F that takes a position in object space and returns a color value. To map from the solid texture space to a point p on a triangle in object space we simply evaluate the solid texture function at p. To collect sufficient information about a triangle, multiple samples must be taken of it. In our implementation we fix the number of samples per triangle side as S. In practice S can vary from 10 to 100, where selection of S is done by the user. A good choice of S depends on the mean size of triangles in the mesh and the frequency content of the solid texture relative to the size of the mesh.

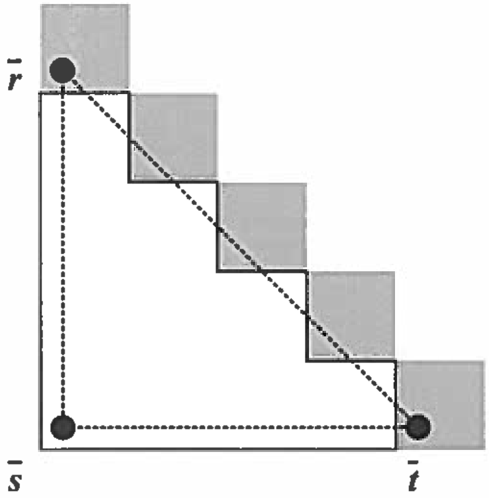


FIGURE 17. The shaded regions may be incorrectly included in the texture generated for the triangle with texture coordinates $\bar{r}$, $\bar{s}$ and $\bar{t}$.

To form a basis for sampling, we randomly select a vertex of the triangle, then treat the two triangle edges emanating from that vertex as the axes of a sampling grid. Each edge is divided S times, forming a grid over the triangle, as shown in Figure 16. Now for each crossing point of the grid, F is evaluated. This generates an isosceles-shaped dataset, where the number of points is

$$\frac{S \times (S+1)}{2} \tag{4.1}$$

Since the data of adjacent triangles are not typically adjacent in the resulting texture map, floating point inaccuracy in the texture lookup functions can result in sampling from distant triangles, as in Figure 17. This produces noticeable artifacts when the mesh is rendered. To eliminate these artifacts we over-scan each triangle by one unit along each axis, resulting in a buffer of valid pixels around the data for each triangle, shown in Figure 18.

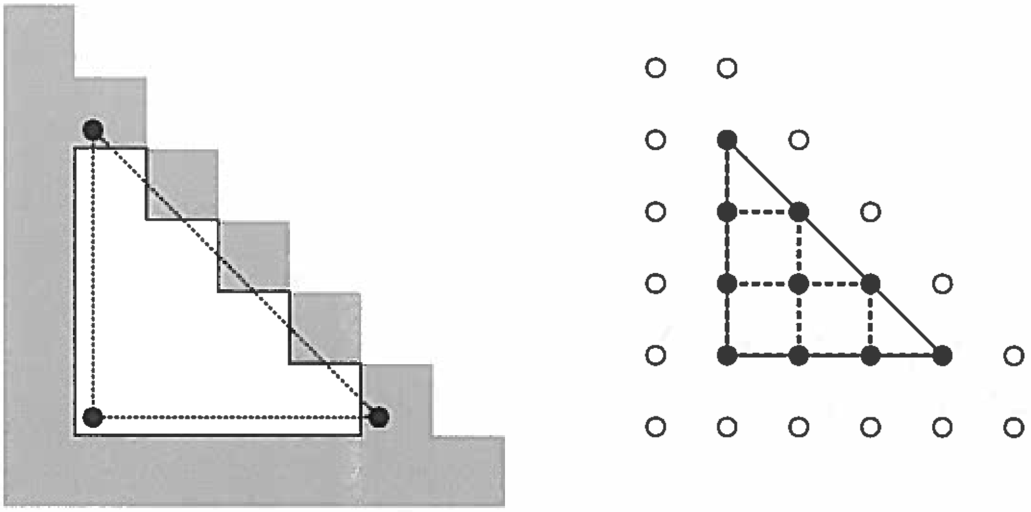


FIGURE 18. The shaded region represents result of over-scanning the triangle. Open circles are the points used for over-scanning the shown triangle.

Texture Packing

Instead of storing the data for each triangle in its own texture map, we pack the data for all of the triangles in the mesh into a single texture map. Because we discretize all triangles uniformly, we can pack them together using a relatively simple packing scheme. Triangles are first packed together along the u texture map axis until reaching the edge of the texture map. Then v is increased by the height of the triangle data. Repeating this process until all triangle data is stored completes the texture map.

In order to pack the data from two triangles along the u coordinate together, we flip one of the data-triangles in both u and v. The inverted data can then be placed adjacent to the first one with no loss of space in the texture map. Rectangles formed by joining two triangles can be packed into a single structure by just placing them adjacent to each other. When the resulting data is written to the texture map, we also record the (u,v) coordinates of each of the three corners of the triangle data and save these coordinates with the mesh vertex data. These corners are one texel inside the data in each dimension, to take advantage of over-scanning. The packing process is illustrated in Figure 19.

More advanced texture packing schemes are discussed by Carr et al. in [1] and Hart et al. in [9]. Carr et al. acknowledge the attractiveness of simple non-adaptive packing schemes similar to the one we have used, but also describe a flaw. This technique performs poorly when the mesh triangles are radically different in size, and when they are irregularly shaped. Both these conditions will cause the data to be sampled irregularly from the texture space, triggering distortion of the texels that are stored in the texture map, as in Figure 20. We overcome this flaw by using sufficiently high sampling rates, and avoiding meshes with irregularly shaped triangles.

Rendering

After sampling the solid texture and packing the resulting texture maps, all that is required to render a mesh with solid texture is traditional texture mapping such as that in *OpenGL*. Each pixel of each rasterized triangle takes its color information from the texture map associated with that triangle. There are a couple of techniques that can make real-time solid rendering look more appealing. We enable linear magnification and minification filtering, which tends to smooth out seams that form along triangle boundaries as well as the appearance of diamond-shaped texels. Gouraud shading also helps hide artifacts along triangle edges.

Results

Here we present the results of our solid texturing algorithm. The images in this section were generated on the same system as specified in Section 2.5. Gouraud shading

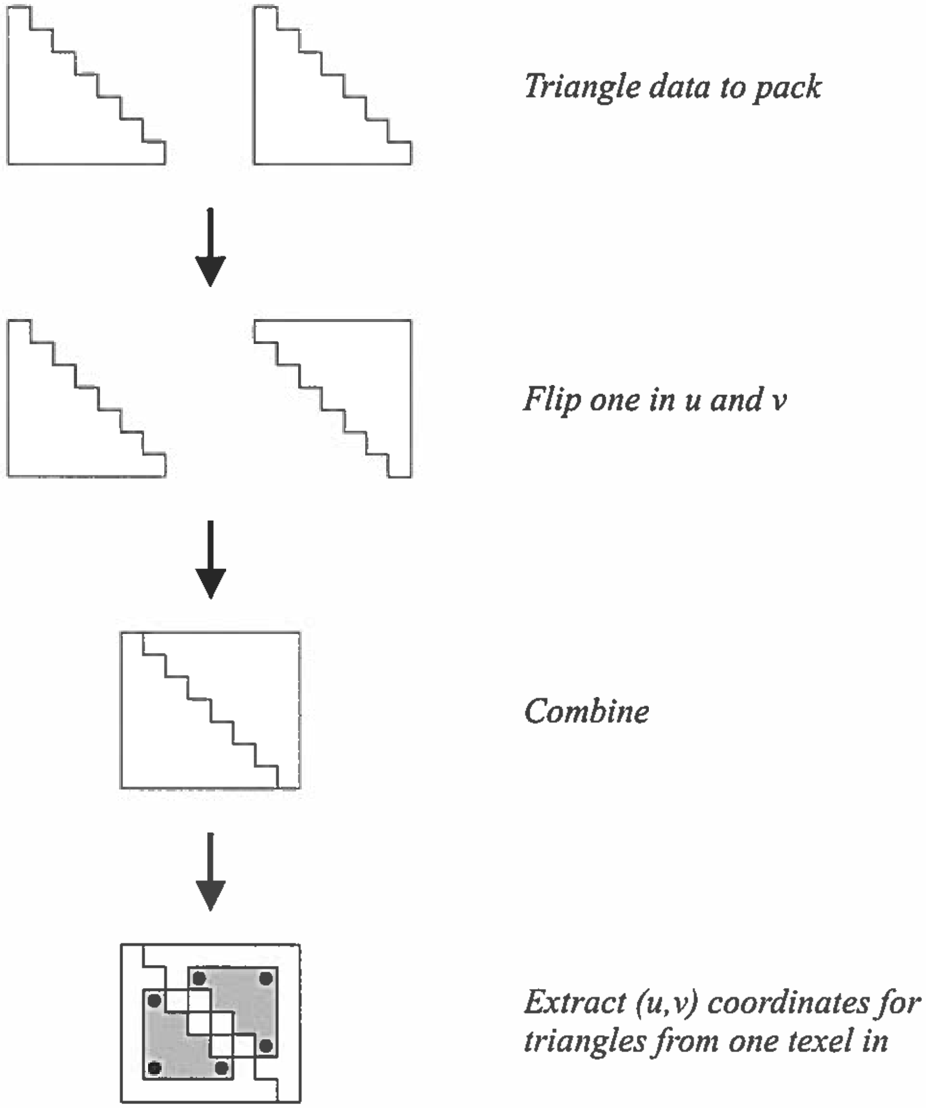


FIGURE 19. Triangles are packed together by inverting every other triangle.

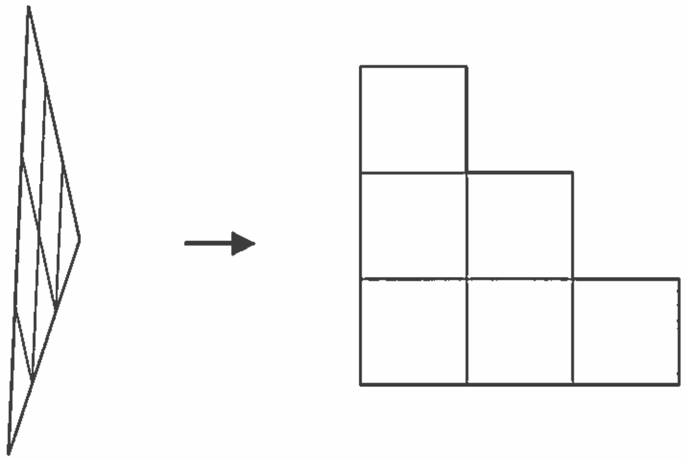


FIGURE 20. The square texels at right, stored in the texture map, get mapped to diamond-shaped regions in the actual triangle.

was enabled for all images, and a light was placed at each camera to illuminate the scene. The following shows the result of varying texture frequency content and the triangle sampling rate. Then we show some examples of correctly rendered meshes.

The images of the Stanford bunny (simplified to 5,000 faces with QSlim) in Figure 21 display the application of a single solid texture while varying the triangle sampling rate. 50 total samples were taken for Figure 21a, 450 for Figure 21b and 3,200 for Figure 21c. In the least-sampled result, artifacts caused by texture mismatches along triangle boundaries are visible. In Figure 21b these artifacts are less prominent, and in the most highly-sampled result, the artifacts are not easily visible.

In Figure 22 we show the result of using a solid texture with a greater proportion of high frequency content. Triangle sampling rates of 50, 450 and 3,200 samples per-triangle were used in the following three images. With a higher-frequency texture, 50 or 450 samples per triangle is not enough to hide sampling artifacts in the textured mesh. At 3,200 samples per triangle, image artifacts are acceptably reduced.

Figure 23 shows a 5,000 face Stanford bunny textured with a solid texture that

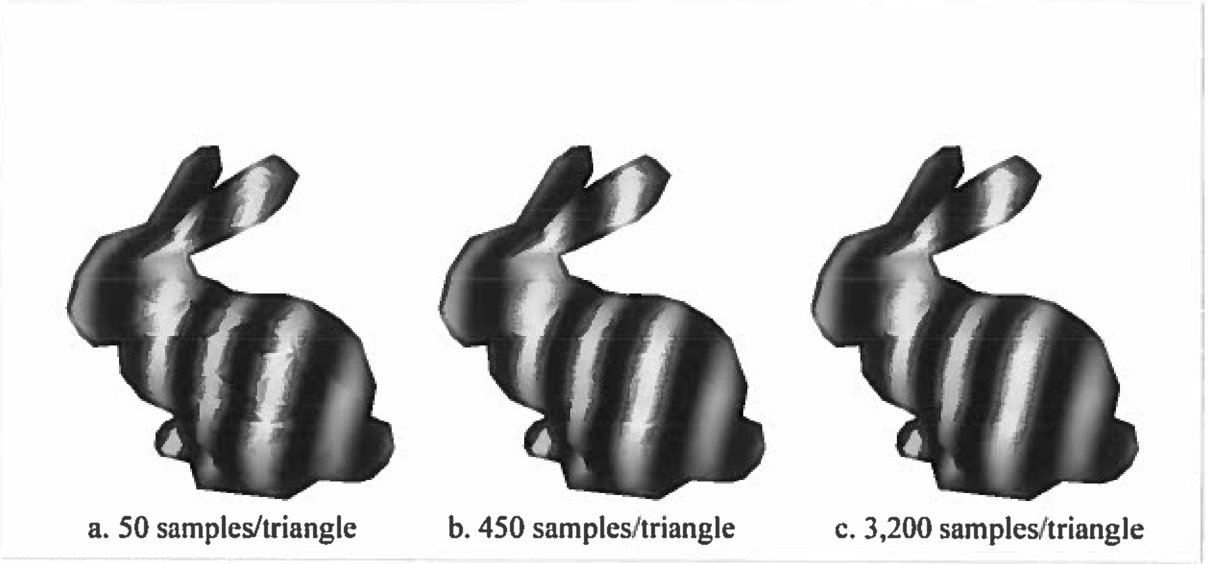

a. 50 samples/triangle b. 450 samples/triangle c. 3,200 samples/triangle

FIGURE 21. Sampling rates of 50, 450 and 3,200 samples per triangle with low frequency texture.

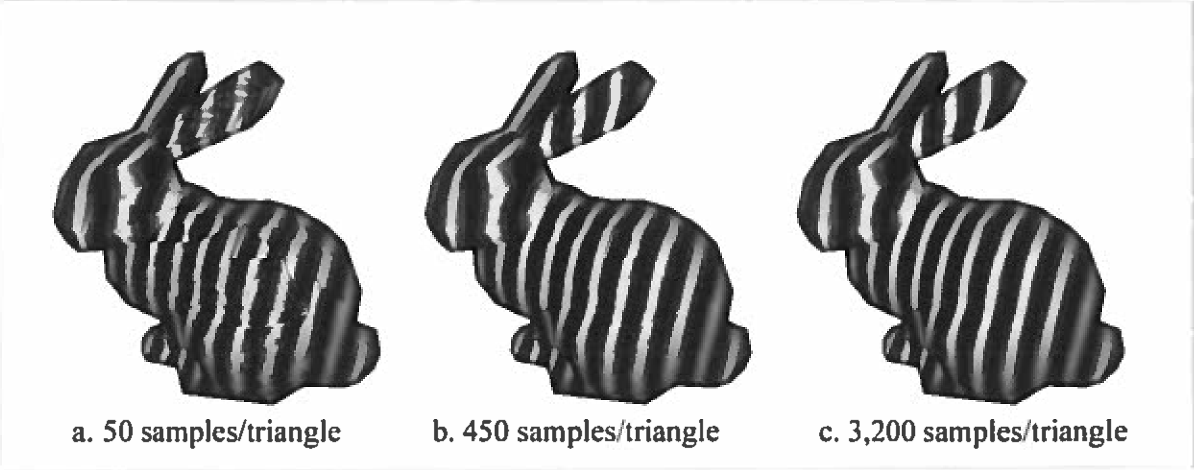

a. 50 samples/triangle b. 450 samples/triangle c. 3,200 samples/triangle

FIGURE 22. Sampling rates of 50, 450 and 3,200 samples per triangle with higher frequency texture.

FIGURE 23. View of the Stanford bunny with marble solid texture.

has a marble appearance. 450 samples per triangle were used, requiring 42 seconds to generate the texture map on the system specified in Section 2.5. Figure 24 shows a close-up of the head of the bunny mesh, displaying the quality of our technique even at short range. Figure 26 and Figure 27 show a different texture applied to the same mesh, again with 450 samples per triangle. The texture maps generated for these meshes were each 2,048 × 2,048 pixels in size, and are shown in Figure 25 and Figure 28.

The above results show that with an appropriate sampling rate, our technique can generate convincing solid textured meshes that can be displayed in real-time. Even using the simplest texture sampling and packing techniques available the results are convincing.

FIGURE 24. Close-up view of the Stanford bunny with marble solid texture.

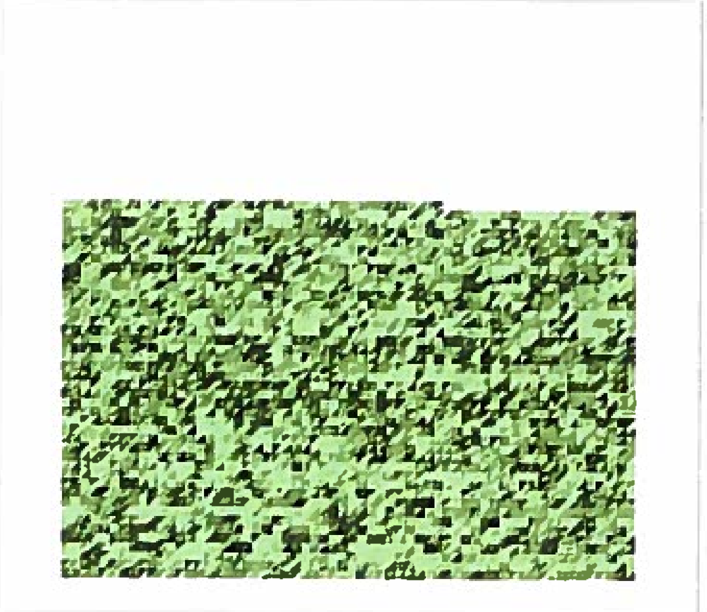


FIGURE 25. The texture map for the Stanford bunny with marble solid texture, originally 2,048 × 2,048 pixels.

FIGURE 26. View of the Stanford bunny with wood solid texture.

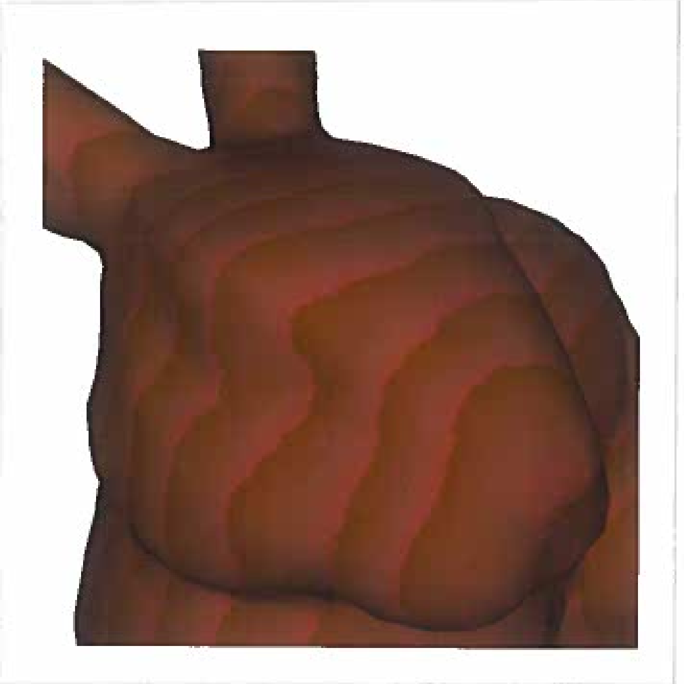


FIGURE 27. Close-up view of the Stanford bunny with wood solid texture.

FIGURE 28. The texture map for the Stanford bunny with wood solid texture, originally 2,048 × 2,048 pixels.

Solid Texturing and Image-driven Simplification

Here we discuss a possible integration of image driven simplification and solid texture. Solid texturing is useful because it can be used to limit texture shifting during simplification by forcing the texture associated with triangles during collapse to be sampled in a well-defined way.

When an edge is collapsed, a set of triangles experience geometric alteration. After collapse these triangles pass though a different region of the solid texturing space. To correctly texture these triangles, they must be re-sampled in the solid texture, and the portions of the texture map that they correspond to must be updated. Since our edge collapse routine never generates new triangles, the texture map can be updated in place. After updating these triangles the texture map for the mesh contains a correct sampling of the solid texture. Hence, edge collapses which introduce very little distortion to the mesh automatically introduce very little distortion to the texture of the mesh. Furthermore, the distortions are predictable, because they are consistent with the original solid texture definition.

CHAPTER V

CONCLUSION

We described an implementation of image-driven simplification which achieves many of the results of Lindstrom and Turk's implementation. Like that of Lindstrom and Turk, our algorithm sacrifices some of the interior surface detail of a mesh in order to more accurately represent silhouette detail. Our method also simplifies away hidden interiors. Our implementation is not nearly as fast as that of Lindstrom and Turk, due to a deliberate lack of optimization.

Our extension of image-driven simplification to a shard-driven approach was shown to produce meshes of acceptable quality relative to the implementation of standard image-driven simplification. This technique allows more freedom in how collapses are considered in an image-driven framework, because shards are rendered and evaluated in isolation from the rest of the mesh. We also demonstrated that iterative image updates for differencing can produce meshes of acceptable quality when using a shard-driven simplification. This allows even further freedom for implementers of image based simplification techniques, since images of the original mesh are not required. Shard-driven simplification was also shown to be much faster than our implementation of image-driven simplification.

We would like to see the shard-driven approach expanded upon, to take advantage of the abstraction available. By considering shards in isolation, we believe fewer cameras can be used than in image-driven simplification. We would also like to see an extension to image-driven simplification in which error for shards is stored in a database, and retrieved upon demand to act as an approximation to the error of similarly-shaped shards elsewhere on the mesh. We believe such caching would result in a significant performance improvement when more advanced and time consuming image difference metrics are used in place of mean squared error.

We also described an implementation of real-time solid texturing in *OpenGL*. By pre-computing the contribution of a three dimensional solid texture to a particular mesh, and storing the contribution in a two dimensional texture map, we were able to achieve interactive frame rates while viewing that mesh with solid texture. While this method makes use of a great deal of texture memory in *OpenGL*, it does not use nearly as much memory as methods that store a fully sampled solid texture. Furthermore, since the contribution is pre-computed, the method is significantly faster than evaluating a solid texture definition in real-time. We proposed using solid texture as an exploratory mechanism for mesh simplification algorithms that use texture. Because the actual texture definition is in the same space as the mesh vertices, texture deviations are less numerous, and more predictable, than those introduced by using traditional two dimensional texture mapping alone.

The possibilities afforded by the image-driven and shard-driven techniques are quite tantalizing. Lindstrom and Turk discussed incorporating models of human vision into the edge collapse heuristic in place of mean squared error. We believe this idea has a great deal of merit. Even the simple addition of models that include the human response to brightness or human threshold sensitivity would allow collapse of geometry that is invisible to a human viewer, freeing polygons to be used elsewhere in the mesh to preserve detail that humans are drawn to. The use of more advanced models, including the Sarnoff and Daly models of human vision, would further increase the possibility of collapsing geometry that contains content invisible to human viewers. We believe the highest quality mesh simplification algorithms will eventually include these kinds of human visual components, with image-driven and shard-driven simplification being the first steps toward an efficient implementation of such algorithms.

BIBLIOGRAPHY

[1] Nate Carr, John C. Hart, and Jerome Maillot. The solid map: Methods for generating a 2-d texture map for solid texturing. http://graphics.cs.uiuc.edu/ jch/papers/papers.html, January 1999. Accessed May 31, 2001.

[2] A. Ciampalini, P. Cignoni, C. Montani, and R. Scopigno. Multiresolution decimation based on global error. *The Visual Computer*, 13(5):228–246, 1997. ISSN 0178-2789.

[3] Wayne O. Cochran, John C. Hart, and Patrick J. Flynn. Fractal volume compression. *IEEE Transactions on Visualization and Computer Graphics*, 2(4), December 1996. ISSN 1077-2626.

[4] Jonathan Cohen, Dinesh Manocha, and Marc Olano. Simplifying polygonal models using successive mappings. *IEEE Visualization '97*, pages 395–402, November 1997. ISBN 0-58113-011-2.

[5] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. *Proceedings of SIGGRAPH 98*, pages 115–122, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[6] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Jr. Frederick P. Brooks, and William Wright. Simplification envelopes. *Proceedings of SIGGRAPH 96*, pages 119–128, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[7] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. *Proceedings of SIGGRAPH 97*, pages 209–216, August 1997. ISBN 0-89791-896-7. Held in Los Angeles, California.

[8] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. *IEEE Visualization '98*, pages 263–270, October 1998. ISBN 0-8186-9176-X.

[9] John C. Hart, Nate Carr, Masaki Kameya, Stephen A. Tibbitts, and Terrance J. Coleman. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 45–53, August 1999. Held in Los Angeles, California.

[10] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. *Proceedings of SIGGRAPH 93*, pages 19–26, August 1993. ISBN 0-201-58889-7. Held in Anaheim, California.

[11] Hugues H. Hoppe. New quadric metric for simplifying meshes with appearance attributes. *IEEE Visualization '99*, pages 59–66, October 1999. ISBN 0-7803-5897-X. Held in San Francisco, California.

[12] Peter Lindstrom. Out-of-core simplification of large polygonal models. *Proceedings of SIGGRAPH 2000*, pages 259–262, July 2000. ISBN 1-58113-208-5.

[13] Peter Lindstrom and Greg Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):98–115, April - June 1999. ISSN 1077-2626.

[14] Peter Lindstrom and Greg Turk. Image-driven simplification. *ACM Transactions on Graphics*, 19(3):204–241, July 2000. ISSN 0730-0301.

[15] Darwyn R. Peachey. Solid texturing of complex surfaces. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):279–286, July 1985. Held in San Francisco, California.

[16] Ken Perlin. An image synthesizer. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):287–296, July 1985. Held in San Francisco, California.

[17] Rémi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, 15(3):67–76, August 1996. ISSN 1067-7055.

[18] Jarek Rossignac and Paul Borrel. Multi-resolution 3d-approximations for rendering complex scenes. Technical Report RC 17697, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598, 1993.

[19] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):65–70, July 1992. ISBN 0-201-51585-7. Held in Chicago, Illinois.