Enhanced Qslim for 3D city model compression and rendering

Mary Knox, Min Ding and Sergey Timofeev

May 14, 2006

University of California, Berkeley
Department of Electrical Engineering and Computer Sciences
Berkeley, CA 94720-1770
# Contents

1 Introduction .................................................. 1
   1.1 Motivation .................................................. 1
   1.2 Overview of Qslim ........................................... 1
   1.3 Organization of this report .............................. 2

2 Solution of the problem .................................... 4
   2.1 General overview ........................................... 4
   2.2 Identify location of the vertices ....................... 4
   2.3 Surface smoothing ........................................ 6
   2.4 Selective edge removal .................................... 8

3 result .......................................................... 11
   3.1 Performance comparison ................................. 11

4 Conclusion and future directions ......................... 15

A Code .................................................................. 16
Chapter 1

Introduction

1.1 Motivation

More and more 3D models are available due to their simplicity to generate. However, the issue with their applications is over-sampled models with too much complexity. Typically, models generated by laser scanner contain more than necessary data points and cause real-time transmission and rendering extremely difficult. In the past, most of the research on mesh compression of 3D models has focused on natural objects. Yet, there is still not a suitable solution to 3D city models, which can efficiently compress aggregate of man-made buildings and preserve the rigidity of these cubes at the same time. As 3D city models become more easily obtainable [1], its applications on city planning and security, etc. urgently require a mesh compression tailored for city models for transmission and rendering.

There has been a significant amount of work devoted to simplify 3D models comprised of triangular meshes. These methods try to reduce the number of vertices describing the object, while keeping its approximate shape. The importance of several approximation constraints was considered while developing these methods: preservation of model topology, boundaries, merging disjoint parts of the object [2]. The Qslim algorithm proposed by Garland and Heckbert [3] is fast and achieves high quality of simplification. In this project, we will enhance this Qslim to make it more suitable for city model compression.

1.2 Overview of Qslim

In this section, we briefly describe Qslim algorithm and demonstrate its deficient performance under high compression ratio of city model. The following overview of Qslim is from [3]. 3D model represented by triangular surfaces is specified by the coordinate of the vertices and the connectivity among the vertices. We define the line joining two vertices as edge.
Pairs of vertices in a mesh that either share the same edge or are disjoint, but within some distance $t$ from each other are first selected. Then each of these pairs is merged into a new vertex; the location of which is chosen to minimize the quadratic error metric (which gives the name to the algorithm). This process is shown on Fig. 1.1. The error metric reflects the distance from the new vertex position to the plane surfaces intersecting at the two old vertices. This algorithm has been shown to provide fast and simple way of compression while resulting in high visual fidelity [2]. While it has been shown to perform well on organic shapes, it causes very noticeable distortion to objects such as buildings, which have straight rigid edges as their primary features. The reason is that Qslim assumes reasonably sparse boundaries [4], which is not valid in city models.

For example, the original city model and the highly compressed model by Qslim are shown in Fig. 1.2. We can clearly see that the buildings in the compressed model lose their rigid box shape. Given the issues with QSLiM for 3D city model compression, we would like to design a new algorithm to accommodate the unique rigidity and simplicity of 3D building meshes. However, we still like to take the advantage of QSLiM’s speed, fidelity and robustness. In this project, we will use the existing QSLiM as a starting point to come up with a compression method to meet the specific requirements of 3D city models.

1.3 Organization of this report

In the next chapter, we will explain in details our algorithm to enhance the existing Qslim. There are three main components in our algorithm, namely vertex location identification, surface smoothing and selective edge contraction. In Chapter 3, we present the compression results with enhanced Qslim and compare them with the compression by Qslim alone. Finally, we conclude this study in Chapter 4.
(a) Original city model with 14243 faces  
(b) Compressed city model with 300 faces

Figure 1.2: The original and compressed city model
Chapter 2

Solution of the problem

2.1 General overview

The issue with Qslim is that it doesn’t distinguish the location of the vertices. For example, the vertices at the corners of the buildings bear much more information about the general shape of the building than the vertices at the ridge or plane surface of the buildings. Therefore, if we can supply the intelligence of identifying the location of the vertices to Qslim and force it to keep the corners vertices to the last for compression, we can obtain an efficient city model compression scheme.

We break our algorithm into three steps to achieve this goal. First, we identify the location of the vertices. It turns out that computing the rank of the normal vector matrix is an efficient and robust way for this purpose. The second step is to smooth the surface where more corner vertices are identified than necessary. This step is to correct the non-ideality from the measurement device and fine details of the buildings which can be corners of window frame, balcony, etc.. Since we are interested in high compression ratio, these fine details are of no interest to us and we would like to remove them. After the above two steps, we classify all the vertices into corner, ridge and plane vertices. Then we force the Qslim to remove the non-corner vertices first. We call this last step selective edge removal. It turns out however, that selective edge removal has an adverse effect under moderate compression. This will be explained in details in Section 2.4.

2.2 Identify location of the vertices

To identify location of the vertices, we first notice that the normal vector of each triangular face indicates the orientation of the surface. We also notice that each vertex has different number of unique normal vectors of the surrounding faces depending on its location. For example, vertex at the corner has three unique normals vectors; vertex at the ridge has
two unique normal vectors and vertex at the plane surface only has one unique normal vector. Therefore, comparison among all the normal vectors of the faces surrounding a vertex promises an effective algorithm to identify the location of that vertex. Therefore, we come up with the following rank test of a normal vector matrix.

Here we assume all the normal vectors are unit vector. The \((i, j)^{th}\) entry of a normal vector matrix for a particular vertex \((v)\) is an logic value indicating whether the normal vectors of \(i^{th}\) and \(j^{th}\) triangular surfaces surrounding \(v\) are orthogonal or not. If orthogonal, the logic value is 1 and 0 otherwise. In practice, we cannot guarantee perfect orthogonality. Instead we define the logic value by comparing two normal vectors’ inner product with a threshold. If the two triangular faces intersect each other with an angle between 30 and 150 degrees, we define them as orthogonal to each other. After determining all the entries of the matrix, its rank readily tells us the location of \(v\).

Fig.2.1 indicates how rank test on normal vector matrix works on vertices at different locations. For simplicity, we assume there are only three triangular faces intersecting at each of all three vertices on the cube. The corresponding normal vector matrix for each vertex along with its rank test result are shown on the right. From this diagram, we can generalize our decision rule to be: plane vertex if rank equals to 0, ridge vertex if rank equals to 2 and corner vertex if rank equals to 3. To accommodate non-ideality of real-world measurement, we also classify vertex as corner if the rank of its normal vector matrix is more than 3. Note that our rank test is robust to any number of faces intersecting at \(v\), since we will just have multiple linearly dependent row/column vectors in this situation.

Some reader might question the computation complexity of this rank test since we have to perform \(n \times (n-1)/2\) orthogonality tests (the diagonal entries are always zeros). Although sequential normal vector grouping is computationally efficient, we will show below that it
is not robust against 3D model from real-world measurement. In 3D city model, we often encounter buildings without rigid corners. One of such examples is shown on the left figure in Fig.2.2. In this case, the yellow surface can intersect red and blue surface with angles below the threshold and thus the normal vector of yellow face is classified non-orthogonal to those of red and blue faces, even though the red and blue faces are orthogonal to each other. If we perform sequential normal vector grouping and accidently start with the yellow surface, we will group the yellow, red and blue surfaces together. We then only have two unique normal vectors and classify this vertex as ridge. However, the rank test can combat this non-ideality as shown on the right figure of Fig.2.2. The rank of 4 indeed tells that there are four unique normal vectors and the vertex will be classified as a corner.

Finally we represent the result of identifying locations of vertices via rank test on normal vector matrix. The left picture in Fig.2.3 shows the original 3D city mesh model, and the right picture shows the model after location classification. We color the corner vertices with blue, ridge vertices with green and plane vertices with red. As shown on the figure, the rank test correctly classifies most of the vertices. However, there are more than necessary corner vertices (blue) identified. In the next section, we show how surface smoothing can remove these extra corner vertices.

2.3 Surface smoothing

When the model data is acquired, the object shapes are sampled discretely in space, which creates distortion to the plane surfaces of the model. In particular, large planes appear as staircase-like structures, with large parallel plane regions separated by short perpendicular segments (see Figure 2.4(a)). The microscopic corners on these uneven surfaces are falsely
identified as building corners by the vertex classification algorithm. Another misclassification problem arises when a true corner is missed by sampling and is “cut off”. For instance, a 90° corner can turn into two 135° corners next to each other, which then may be mistaken for two ridge vertices. Setting a lower threshold in the previous step on the maximum allowable angle between “parallel” surfaces will result in detection of two corners, which is again a mistake. We present a method to smoothen the surfaces and thus make vertex classification descriptive of the overall building shape.

In both situations described above the erroneous section of the model has two corner or ridge vertices placed close to each other. These pairs of vertices can be merged into one by removing the connecting edge. So first we need to specify the set of all edges to be removed. An edge is specified by listing the two endpoint vertices. Then the model is compressed by Qslim, with the option of listing the edges to be removed first. An outline of the processing steps is given below:

1. Go through all vertices that have been classified as corner or ridge,
2. For each of these vertices assemble a list of corner/ridge vertices it is connected to,
3. Compare the length of each connecting edge to a distance threshold: if an edge length is less than the threshold, add this edge to the list of the ones to be removed,
4. Run Qslim compression with the specified edge set, with face number reduction less than the number of removed edges,
5. Run vertex classification on the resulting model; if there are still unwanted mini-edges left, repeat the procedure from step 1.

Even though Qslim allows to contract specific edges, the main compression parameter is still the number of faces in the resulting model. Thus if Qslim has compressed all the specified edges, but the result hasn’t reached the target number of faces, Qslim keeps on compressing other edges. We have found this to cause significant errors in the models, therefore the above iterative procedure of removing a few edges at a time has performed better. The results of surface smoothing algorithm are presented in Figure 2.4.

![Figure 2.4: Smoothing of jagged surfaces](image)

2.4 Selective edge removal

In order to preserve the rigidity of buildings, we specified the edges for Qslim to contract. The edges which pertain to the corners of buildings are the most important edges to keep. By keeping them, the outline of buildings will be maintained and thus the buildings will retain a boxlike structure as desired.

To simplify the model we use the Qslim option, described in Section 2.3, that allows the user to specify the edges to contract. We use the properties of the vertices (see Section 2.2) in an edge to determine the edges to contract. We wanted to contract any edge in which one of the vertices was a plane vertex or both of the vertices were ridge vertices. Another way of describing our algorithm is that we only kept edges connecting a corner vertex to a ridge vertex and edges connecting two corner vertices. Doing so minimizes the number of
edges needed to keep the corners in tact. In an ideal situation we will only keep three black edges as shown in Figure 2.5.

Once the edges to contract are specified, we run Qslim to reduce the number of faces of the original by the number of edges we have specified. We then continue to use Qslim without the selective edge removal to further simplify the model.

The results from using Qslim with and without selective edge contraction are shown in Figure 2.4. Note that we have added the mesh in order to see the number of faces used in the model. After using selective edge removal the buildings are represented with fewer faces. Though the selective edge removal reduce the buildings to fewer faces, recall that the total number of faces for the two models are the same. Thus, the model using selective edge removal distributes the rest of the faces to smaller objects/features. An example of this is shown in Figure 2.4. Note the areas circled in red are represented with far more faces for the model using selective edge removal than the model without selective edge removal. Furthermore, the model without selective edge removal is more visually descriptive of the shapes of these small objects/features. For these reasons, we chose not to pursue selective edge removal in our comparison between our model and the model using Qslim only. We define the Qslim combined with our vertex location identification and surface smoothing as enhanced Qslim from now on.

Figure 2.5: Ideal Edges to Keep to Preserve Corner
Figure 2.6: Results after applying Qslim to reduce an original model with 14485 faces to 3578 faces

Figure 2.7: Zoomed in version of Figure 2.4
Chapter 3

result

3.1 Performance comparison

In Figure 3.1, we have added texture to the original model (14485 faces) as well as a simplified model with 300 faces. After adding texture, it is easier to recognize buildings in the model. However, the differences in the shapes of the building, which can be seen in Figures 3.1(b) and 3.1(d), is not as noticeable after the texture is added. For this reason, we included pictures without texture. We also applied our algorithm to a larger model with 69070 faces and found the results to be consistent with those shown in the following figures. For ease of viewing we chose to include the models with fewer buildings.

From Section 2.4, we concluded that it was not worthwhile to remove selective edges. Thus the following results will compare models using Surface Smoothing in addition to Qslim (Enhanced Qslim) with models that only use Qslim to compress it.

Figure 3.1 shows the Enhanced Qslim and Qslim models after compressing the model to 500 faces from 14485 faces. From the figure, we see that the Enhanced Qslim model better retains the structure of the buildings. For example, in Figure 3.2(a) the bottom corner of the building lies directly below the top corner of the building; whereas in Figure 3.2(b) the bottom corner has shifted outside the outline of the building. Comparing Figures 3.2(c) and 3.2(d), we notice that the Enhanced Qslim model does a better job of keeping the ridges of the building straight. Moreover, we compared the number of faces to represent some of the buildings in the model. The buildings for which we counted the faces are shown in Figure 3.1. The results are shown in Table 3.1. For the models with 500 faces the Enhanced Qslim model used less faces to define the buildings than the Qslim model. In this highly compressed case, having fewer faces on buildings can be a benefit because the entire ridge of the buildings is more likely to be straight.

When we continued to compress the model further to 300 faces, the model lost much of its detail and it is difficult to tell the difference between the Enhanced Qslim and Qslim
Figure 3.1: Models with and without texture

Table 3.1: Comparison of face numbers between Qslim and enhanced Qslim
Figure 3.2: Enhanced Qslim and Qslim models with 500 faces

(a) Building 1 in Enhanced Qslim model

(b) Building 1 in Qslim model

(c) Building 4 in Enhanced Qslim model
models. Similarly from Table 3.1, we see that the number of faces representing each building is very similar. Thus for extreme compression, applying our algorithm does not provide more benefit than using Qslim alone. Furthermore, at these high compression levels, the larger buildings maintain their boxlike structure. However, the smaller buildings and houses in the model collapse.
Chapter 4

Conclusion and future directions

A modification to the existing Qslim compression software was successfully implemented. The proposed modification improves the appearance of city models under moderate to high compression by preserving box-like shape and straight linear borders of the buildings in the models. The algorithm was tested on several models and was shown to be scalable.

Apart from showing that enhanced Qslim produces better representation under high compression ratio, we believe that we can take advantage of knowing coordinate of the corner vertices. In particular, the corner vertices should be sufficient information to produce general shape of buildings resulting even higher compression ratio with the rigidity intact. However, this requires us to modify the internal structure of Qslim. If more time is allowed in the future, we plan to proceed in this direction and expect to obtain better results.
Appendix A

Code

% main.m
% Controller program, calls other functions

[v,f] = load_data('half_temp8.smf',55798);
th = 2^(-1); % angle less than 30 is considered coplanar
for i = 1:size(v,1)
    NV = normal_vectors(i,v,f);
    switch group(NV, th)
        case 0
            c(i) = 1;
        case 1
            c(i) = 2;
        case 2
            c(i) = 2;
        case 3
            c(i) = 3;
        otherwise
            c(i) = 3;
    end
end
e = select_edges(v,f,c,3);
output_smf_e(v,f,c,e,'half_temp9.smf');

% load_data.m
% reads data from .smf file

function [v,f] = load_data(filename,m)

% m - number of lines in the file to read
% v - array of vertices
% f - array of faces

fid=fopen(filename,'r');
data = fscanf(fid,'%1c %g %g %g ',[m,4]); %Scan all data
data = data(:);
data_new = zeros(m,4);
%Break into m x 4 matrix
counter = 1;
for m_count = 1:m
    for n_count = 1:4
        data_new(m_count,n_count) = data(counter);
        counter = counter+1;
    end
end

%Finding when it changes from v to f
n = 1;
while data_new(n,1) == 118;
    n = n+1;
end

change_v_to_f = n; %Split up v and f values and save separately
v = data_new(1:change_v_to_f-1,2:4);
f = data_new(change_v_to_f:m,2:4);

fclose(fid);

________________________________________

% normal_vectors.m
% Finds normal vectors to the faces adjacent to the given vertex

function N = normal_vectors(vertex,v,f)
% vertex - coordinates of the vertex under consideration
% v - array of vertices
% f - array of faces

[m_v,n_v] = size(v);
[m_f,n_f] = size(f);

r = min(abs(f - vertex)'); %for rows with vertex, r = 0
keep_face = zeros(1,3);
for row_number = 1:length(r)
    if r(row_number) == 0
        keep_face = [keep_face;f(row_number,:)];
    end
end

[m_k,n_k] = size(keep_face);
keep_face = keep_face(2:m_k,:);
%Matrix with rows containing the vertex chosen

for row = 1:m_k-1
    v1 = v(keep_face(row,1),:) - v(keep_face(row,2),:);
    v2 = v(keep_face(row,2),:) - v(keep_face(row,3),:);
    N(row,:) = cross(v1,v2)/norm(cross(v1,v2));
    % Normal value for each face row which contained vertex
end

if m_k < 2 %in case we cannot identify this vertex from the face array
    N = 0;
end

% group.m
% Determines if a given vertex is a corner, on a ridge, or on a plane

function output = group(N, th)

% N - normal vectors corresponding to faces adjacent to the vertex
% th - threshold on the angle between faces to decide if they are parallel

num = size(N,1);
indicator = ones(num);
for i = 1:num
    for j = 1:num
        if abs(dot(N(i,:),N(j,:))) >= th
            indicator(i,j) = 0;
        end
    end
end
output = rank(indicator);

% select_edges.m
% Select a set of short edges for smoothing

function e = select_edges(v,f,c,th)
% v - array of vertices
% f - array of faces
% c - array of colors, one per vertex
% th - distance threshold to decide if the edge is short
% e - list of edges that should be removed

e = [];

for k = 1:size(v,1)
    if c(k)>=2
        x = v(k,:);
        adj_corners = [k]; % the first in adjacent corners list is itself
        for k2 = k:size(f,1)
            if ~isempty(find(f(k2,:)==k)) % if the face has this vertex
                for k3 = 1:size(f,2)
                    if (isempty(find(adj_corners==f(k2,k3)))) & (c(f(k2,k3))>=2) % if the vertex is not already in the adjacent list, % and is indeed a corner/edge
                        adj_corners = [adj_corners,f(k2,k3)];
                    end
                end
            end
        end
    end
end
if length(adj_corners)>1
    adj_corners = adj_corners(2:length(adj_corners));
    adj_coords = v(adj_corners,:);
    dist = zeros(1,length(adj_corners));
    for k2 = 1:length(dist)
        dist(k2) = sqrt(sum((adj_coords(k2,:)-x).^2));
    end
    t = find(dist<=th);
    if ~isempty(t)
        c(adj_corners(t))=0; % flag that this vertex has been processed
        c(k)=2;
        for k2 = t
            e = [e;k,adj_corners(k2)];
        end
    end
end
end

% edge_con.m
% Selects the edges to be removed during compression (stage 3 of the algorithm)
% Edges which are on the plane or on a ridge will be listed

function e = edge_con(f,c)

% c - array of colors
% f - array of faces

[m_f,n_f] = size(f);
e = zeros(1,2);
n = 1;

for row = 1:m_f
    v1_row = f(row,1);
    v2_row = f(row,2);
    v3_row = f(row,3);
    if c(v1_row) == 1 | c(v2_row) == 1 | (c(v1_row) == 2 & c(v2_row) == 2)
\[ e(n,:) = [v1\_row \ v2\_row]; \]
\[ n = n+1; \]
\[ \text{elseif } c(v3\_row) == 1 | c(v2\_row) == 1 | (c(v3\_row) == 2 \& c(v2\_row) == 2) \]
\[ e(n,:) = [v2\_row \ v3\_row]; \]
\[ n = n+1; \]
\[ \text{elseif } c(v1\_row) == 1 | c(v3\_row) == 1 | (c(v1\_row) == 2 \& c(v3\_row) == 2) \]
\[ e(n,:) = [v1\_row \ v3\_row]; \]
\[ n = n+1; \]
end
end

% output_smf.m
% Writes an smf file with vertex colors and lists the edges to contract first

function output_smf(v,f,c,e,filename)

% v - array of vertices
% f - array of faces
% c - array of colors
% e - array of edges

fid = fopen(filename,'w');
[m_v,n_v] = size(v);
[m_f,n_f] = size(f);
[m_e,n_e] = size(e);

fprintf(fid,'v %g %g %g\r',v(1,:));
for row = 2:m_v
    fprintf(fid,'
v %g %g %g\r',v(row,:));
end
for row = 1:m_f
    fprintf(fid,'f %g %g %g\r',f(row,:));
end
fprintf(fid,'bind c vertex\r');
for row = 1:m_v
    if c(row) == 1
        fprintf(fid,'\nc 1 0 0\r');
    elseif c(row) == 2
        fprintf(fid,'\nc 2 0 0\r');
    end
end

fprintf(fid,'\nc 0 1 0\r');
else
    fprintf(fid,'\nc 0 0 1\r');
end
for row = 1:m_e
    fprintf(fid,'\ne %g %g\r',e(row,:));
end
fprintf(fid,'\n');
fclose(fid);

