Joint Connectivity Compression and Mesh Rendering

Gang Lin, *member, IEEE*, Wenji Zhang

Abstract **-- An efficient algorithm for joint connectivity compression and fast rendering of triangle meshes is proposed in this paper. We introduce an ordering scheme for traversing the vertices in the mesh such that the connectivity information of a large proportion of faces can be captured with only very few bits. This scheme is simple and easy to implement, and yields comparable compression performance when compared to some of the reported methods. Moreover, the proposed compression scheme facilitates the efficient rendering of the mesh. Based on our reorganized representation of the mesh, we present an efficient algorithm to achieve so called Minimum Time Rendering (MTR) by using a relatively small FIFO mesh buffer. The order of MTR rendering sequence is exactly same as the order of reordered faces of the mesh, thus the MTR rendering does not cost any additional computation. Moreover, due to the well predictable order of the vertices in mesh buffer, the proposed MTR algorithm avoids searching the buffer so as to access certain vertex data, which speeds up the mesh rendering.**

Index Terms **-- Triangle mesh, Connectivity compression, Adjacency list, Minimum-time rendering**

I. INTRODUCTION

Polygonal model plays an increasingly important role in medical imaging, manufacturing, video games, Geographic Information Systems, scientific visualization and industrial CAD applications. With the advent of range scanning systems, meshes of extreme complexity are rapidly becoming commonplace. These large meshes are very expensive to store and transmit. Therefore, more and more attentions have been drawn on the field of mesh compression, and a number of compression scheme have been proposed [1, 2, 3, 4, 5].

Generally, the computer graphics rendering pipeline consists of three conceptual stages: application (e.g., feeding commands to the graphics subsystem), geometry subsystem (i.e., per-polygon operations, such as coordinate transformations, lighting and clipping.) and raster subsystem (i.e., per-pixel operations, such as writing color values into the frame buffer, depth buffering and alpha blending.). When handling some large meshes, the pipeline stage that does geometry operations is heavily loaded, and consequently real time graphics hardware is increasingly facing a memory bus bandwidth bottleneck in which the large amount of geometry data cannot be sent fast enough to graphics pipeline for rendering due to slow memory subsystems. Therefore, it is quite desired that a mesh compression technique facilitate the fast rendering of the mesh.

In this paper, we propose an efficient algorithm for joint connectivity compression and rendering of the triangle meshes. Section II describes the mesh compression algorithm. Based on our representation of the mesh, we present an efficient rendering scheme in section III. Section IV shows the experiment results for some typical 3D meshes. Finally we give our conclusions.

II. COMPRESSION ALGORITHM

VRML [10] is the emerging standard for the delivery of 3D content in a networked environment. In VRML standard, a polygon model with $|V|$ vertices and $|F|$ faces is generally represented by a vertex coordinates array V, a face array F and one or more optional property array such as surface normal, color and/or texture coordinates.

A. Vertex reordering

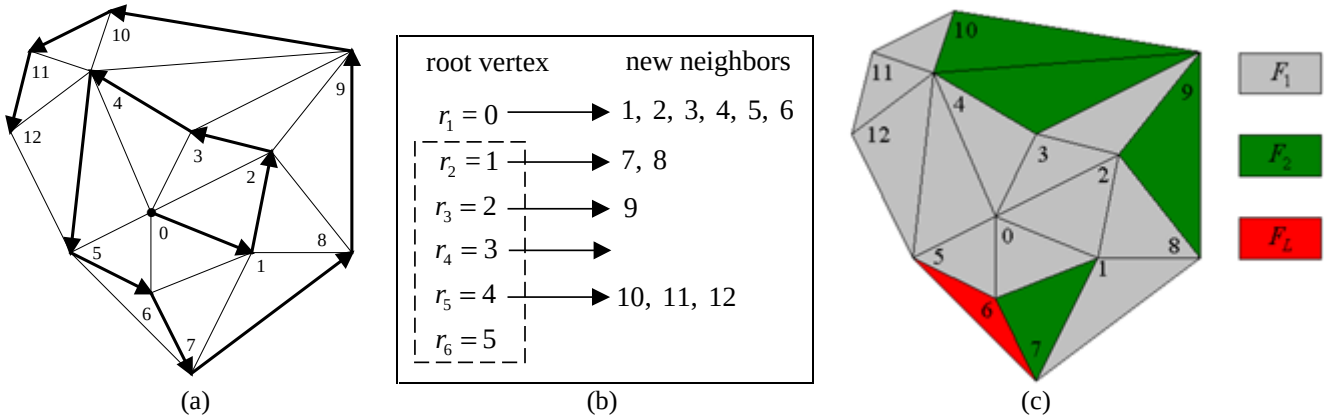
In the following, we propose a scheme for reordering the vertices in a mesh such that in this ordering a good proportion of the connectivity information is captured. The proposed vertex reordering has the following basic structure: We maintain a list R that is gradually updated until it includes all the vertices in the mesh, and we call R the "reordered vertex array". The algorithm also produces a set of lists of the form $L_i: r_i \to \{w_1^i, w_2^i, \cdots, w_{l_i}^i\}$, $1 \le i \le K$; which we refer to as "trimmed adjacency lists", such that as the algorithm terminates R is exactly: $\{r_1, w_1^1, \cdots, w_{l_1}^1, \cdots, w_1^i, \cdots, w_{l_i}^i, \cdots, w_1^K, \cdots, w_{l_K}^K\}$.

In detail, it works as follows. We pick an arbitrary initial vertex r_1 in V and set $R=\{r_1\}$. Set ROOT= r_1. Let w_1^1 be an arbitrary neighbor of r_1 and define the first list L_1 to be $L_1: r_1 \to \{w_1^1, w_2^1, \cdots, w_{l_1}^1\}$, where $w_1^1, \cdots, w_{l_1}^1$ are decided by starting from w_1^1 and enumerating the neighbors of r_1 in a clockwise or counter-clockwise fashion that depends on the input mesh. Notice that in the above notation l_1 is exactly the valence of root r_1, i.e., $l_1 = \text{valence}(r_1)$. We then update R by appending to it $w_1^1, \cdots, w_{l_1}^1$. Thus, at this stage of the algorithm, $R=\{r_1, w_1^1, \cdots, w_{l_1}^1\}$.

We proceed as follows to build $L_2, L_3, \cdots, L_K$ and update R as each list is built.

- Set $i=1$
- While $|R|<|V|$,
  - $\Rightarrow$ $i=i+1$
  - $\Rightarrow$ Set ROOT to be the vertex right after its current position in R;
  - $\Rightarrow$ Set r_i = ROOT. (For example, if $i=2$, then $r_2 = w_1^1$ in the above notation.)
  - $\Rightarrow$ Set L_i: $r_i \to \{w_1^i, w_2^i, \cdots, w_{l_i}^i\}$, where L_i consists of exactly the neighbors of r_i in the mesh *which have not appeared in* R. As in L_1, the order of vertices in L_i is decided by enumeration in the same orientation. (The vertex w_1^i has yet to be specified, but we omit the details here.) Note: Unlike for $i=1$, $l_i <$ valence(r_i) and can even be zero (in this case, L_i is an empty list).
  - $\Rightarrow$ Update R by appending $w_1^i, \cdots w_{l_i}^i$ to its end.
- End while
- Set root r_{i+1} to be the vertex in R right after r_i.

Note that r_1 and w_1^1 can be chosen arbitrarily, but once they are fixed, then the ordering is determined uniquely. Fig.1 (a) shows how the vertices are traversed in a small mesh. In this example, $|V|=13$, $K=5$. Trimmed adjacency lists L_i we generate are shown in Fig.1 (b), and the final reordered vertex array is $R=\{0,1,2,\cdots,12\}$. We can see that the list with root $r_4=3$ is empty.



(a) (b) (c)

Fig.1 (a) Vertex traversing and reordering; (b) Trimmed Adjacency lists; (c) Connectivity encoding.

Fig.3 shows the traversal order of the Venus model (black solid line). The blue dot is the initial vertex r_1. Notice that while the order of traversal does not necessarily form a path on the graph of the mesh, a large proportion (99.7% in this example) of adjacent vertices $v_i v_{i+1}$ indeed form edges of the underlying graph.

B. Connectivity encoding

We compress the connectivity of the mesh exactly at the same time of the above vertex reordering process. By definition, each vertex in the original mesh appears in the reordered vertex array R exactly once, as shown in Fig.2 (a), where r_1 is the initial vertex, W_i is the trimmed adjacency list with root r_i, thus in this notation $R=\{r_1, W_1, W_2, \cdots, W_K\}$. Fig.2 (b) shows the corresponding code bits we generate.

Every vertex in $\bigcup_{i=1}^{K} W_i$ is assigned a bit of type I and II. For notational convenience, we are going to denote the type I and II bits associated with w_j^i by $a_0(w_j^i)$ and $a_1(w_j^i)$ respectively. The type III and IV bits associated with r_i, $2 \le i \le K$ are denoted by $a_2(r_i)$ and $a_3(r_i)$ respectively.

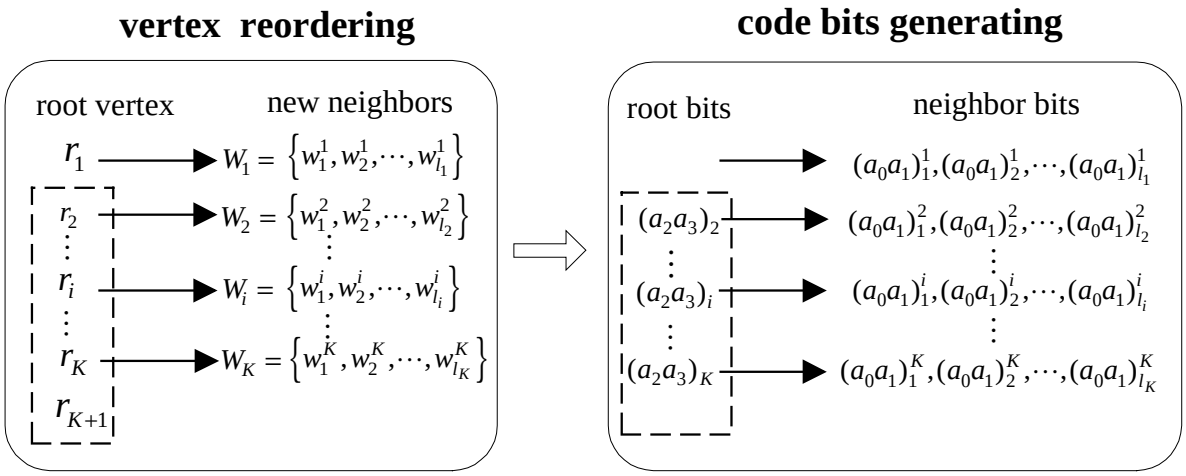


Fig.2. This figure shows the code bits generated by our compression algorithm.

1) Type I bits

The bit $a_0(w_j^i)$ records whether w_j^i is the last vertex in the list W_i.

2) Type II bits

We need to distinguish two cases: $j < l_i$ and $j = l_i$.

- [$j < l_i$]: $a_1(w_j^i)$ records whether $\{r_i, w_j^i, w_{j+1}^i\}$ is a face in the input mesh. For example, in Fig.1 the face *(1,7,8)* is in this form, so $a_1(7)$ is set to 0. Note that $\{r_i, w_j^i, w_{j+1}^i\}$ doesn't necessarily form a face, although it doesn't happen in the example mesh used in Fig.1. Empirical observation suggests that most tuples $\{r_i, w_j^i, w_{j+1}^i\}$(more than 98.7% in all our test cases) are indeed faces in the input mesh, so in our convention most of $a_1(w_j^i)$ are set to zero and hence a simple entropy coding on the bit stream $(a_1(w_j^i))_{j=1,\cdots,l_i-1;\, i=1,\cdots,K}$ achieves good compression.
- [$j = l_i$]: $a_1(w_j^i)$ records whether $\{r_i, w_{l_i}^i, r_{i+1}\}$ is a face in the input mesh. For example, the face *(1,8,2)* has this form, so in Fig.1 $a_1(8) = 0$. As in the previous case, most tuples $\{r_i, w_{l_i}^i, r_{i+1}\}$ (more than 88.5% in all our test cases) are found to be actual faces in the mesh.

For later reference, the faces captured by bits $a_1(w_j^i)$ are denoted by F_1, as shown by gray faces in Fig.1 (c).

3) Type III bits

$a_2(r_i)$ records whether the list W_i is empty or not. For example, in Fig.1, $a_2(1) = a_2(2) = 0$, $a_2(3) = 1$.

4) Type IV bits

Let $\mathrm{NEXT}(r_i) = \begin{cases} w_1^i & \text{if } W_i \neq \varnothing \\ r_{i+1} & \text{if } W_i = \varnothing \end{cases}$, $\mathrm{PREV}(r_i) = w_{l_{i'}}^{i'}$ where $i' < i$ is the largest index such that $w_{i'}$ is nonempty. Then $a_3(r_i)$ records if $\{r_i, \mathrm{PREV}(r_i), \mathrm{NEXT}(r_i)\}$ forms a face in the mesh. In Fig.1, *(1,6,7), (3,9,4) and (4,9,10)* are all examples of faces of this form. Unfortunately, there is no non-example in the mesh used in Fig.1, but in other more complicated mesh we have tested, $\{r_i, \mathrm{PREV}(r_i), \mathrm{NEXT}(r_i)\}$ can fail to be a face, but very rarely. Again, based on this empirical observation, entropy coding applied to $(a_3(r_i))_{i=2,\cdots,K}$ yields good compression. All faces in above form are denoted by F_2, as shown by green faces in Fig.1 (c).

5) Handling other faces

At each vertex list L_i, after encoding the faces described above, we check if there still exist any "left faces" which have not been encoded, and the root r_i is one of its bounding vertices. We denote left face at r_i as the form $\{r_i, m_{j,1}^i, m_{j,2}^i\}$, $1 \le j \le J$, where J

is the number of left faces at r_i. To encode these left faces, we record the number J and the two difference values $\delta^i_{j,1} = m^i_{j,1} - r_i$ and $\delta^i_{j,2} = m^i_{j,2} - r_i$ for each face. Because after traversing the vertices in adjacency-list order, three vertex indices of each face in F are very close, we can reduce code bits by encoding the difference pairs $\delta^i_{j,1}$ and $\delta^i_{j,2}$. For example, the face *(5, 7, 6)* in Fig.1 is a left face, we record two difference values are $\delta^5_{1,1} = 7-5 = 2$ and $\delta^5_{1,2} = 6-5 = 1$ respectively. We denote all left faces as $F_L = F \setminus (F_1 \bigcup F_2)$, as shown by red face in Fig.1 (c).

At the decoder, the decompression algorithm reads the compressed representation from the input bit stream and enumerates the face set F_1, F_2 and F_L. Table.2 shows the decoded faces and their corresponding membership for the simple mesh in Fig.1. Notice that in comparison with original format of the input mesh, the decoded mesh have the following properties: (1) All faces are reorganized in the increasing order with regard to their first vertex indices; (2) For each face, the first index is the smallest among three indices of its bounding vertices. In the following sections, we will show that above properties are beneficial to MTR rendering of the mesh.

III. MINIMUM TIME RENDERING

In the traditional polygon-rendering pipeline, bus traffic between the graphics subsystem and memory can become a bottleneck when rendering geometrically complex meshes. In a triangle mesh, each triangle is typically represented by its three vertices, where each vertex contains three coordinates specified by floating point numbers, and possibly some other components such as color, normal and texture coordinates. Therefore the vertex data transmission is expensive, and it is desirable to minimize the number of times this data must be loaded into graphics pipeline from memory during rendering the mesh.

The traditional wisdom of efficient rendering occurs to be based primarily on the idea of triangle strip [7, 8]. The basic observation is that a long strip of triangles can be rendered roughly 3 times more efficient (in terms of the amount geometry information needed to be sent through the graphics pipeline) than a straightforward scheme based on triangle-by-triangle rendering. Implementing such triangle strips require a set of three vertex registers in the graphics processor. The use of larger vertex register set has the potential to further reduce geometry bandwidth by another factor of nearly 2. Deering [6] suggests the use of an on-board vertex buffer of sixteen vertices. The input mesh is represented as a stream of variable length instructions that load vertices into the buffer and use buffer entries to form generalized triangle strips. Bar-Yehuda and Gotsman [9] proposed a MTR rendering algorithm such that each vertex of the mesh is sent through graphics pipeline only once by using graph separating algorithms combined with divide-and-conquer strategy. They show that to render an arbitrary mesh with $|V|$ vertices, a buffer size $C\sqrt{|V|}$ is both necessary and sufficient, where C is between (1.649, 12.72).

In order to reduce the vertex data sent to graphics pipeline during rendering, the key is to reorder the faces within the mesh so as to maximize references to vertices already loaded in buffer. As described in previous section, our proposed compression scheme traverses all vertices of the input mesh in a very compact order, i.e., adjacency lists, and the vertex indices of each face change accordingly with respect to this new vertex order. Moreover, all faces are reorganized into a well predictable order, as described by properties (1) and (2) in previous section. These properties facilitate MTR rendering of the mesh. In the following, we describe our MTR rendering algorithm for mesh $M(V, F)$, where V and F are the reordered vertex list and reordered face list.

A. Algorithm

ALGORITHM: MTR

Description.

Rendering the reorganized mesh in minimum time.

Input.

V -- vertex list of mesh M, i.e., $V = \{0, 1, \cdots, |V|\text{-}1\}$; F -- face list of mesh M.

Result.

Rendering sequence of M, and the required buffer size b.

Internal variables.

V_b: vertices in buffer

$F'(v) = \{ f \mid f \in F,\ v \text{ is the first vertices of } f,\ \text{i.e., } f \text{ has the form } (v, \cdot, \cdot) \}$

Procedure $\mathrm{MTR}(V,F)$:

1. $V_b=\varnothing,\ b=0,\ i=0$
2. While $F\neq\varnothing$
3. $v_{focus}=i$ /* pick a focus vertex sequentially */
4. If $v_{focus}\notin V_b$
5. **push(**v_{focus}**),** $V_b=V_b\cup\{v_{focus}\},\ V=V\setminus\{v_{focus}\}$
6. Let $v_{\max}=\max_{v}\{\,v\mid v \text{ is the bounding vertex of } f\in F'(v_{focus})\,\}$
7. If $v_{\max}>\max\{V_b\}$, set $V^*=\{\max\{V_b\}+1,\max\{V_b\}+2,\cdots,v_{\max}\}$
8. **push(**V^***),** $V_b=V_b\cup V^*,\ V=V\setminus V^*$ /* vertices in V^* are needed for rendering $F'(v_{focus})$ */
9. If $b<|V_b|$, then $b=|V_b|$ /* update the required buffer size */
10. Render $F'(v_{focus})$
11. $F=F\setminus F'(v_{focus})$ /* faces in $F'(v_{focus})$ have been rendered */
12. **pop(**v_{focus}**),** $V_b=V_b\setminus\{v_{focus}\}$ /* v_{focus} will not be needed any more */
13. $i=i+1$

B. Correctness and related properties

[I] The above algorithm always terminates after no more than $|V|$ steps: at each stage, we pick one vertex as v_{focus}, and render $F'(v_{focus})$. Therefore, after every vertex in $V=\{0,1,\cdots,|V|-1\}$ has become v_{focus}, all faces in F will be rendered, thus $F=\varnothing$.

[II] Each vertex is sent to mesh buffer only once: at each stage, once $F'(v_{focus})$ has been rendered, v_{focus} will be popped out of the buffer. Since this point, v_{focus} will not be needed any more (due to the properties (1) and (2) described in previous section), thus v_{focus} will never be sent to buffer again.

[III] The vertices in V_b are exactly in increasing order (with regard to their indices): This is an immediate consequence of step 8. This property implies that we do not need to search the buffer to access certain vertex data during the above procedure, which speeds up our MTR rendering.

[IV] The buffer is served as a FIFO buffer: it is due to [III] and the fact that each vertex in $V=\{0,1,\cdots,|V|-1\}$ becomes v_{focus} sequentially.

[V] The order of rendering sequence is exactly same as the order of the faces in F: This can be easily derived from [IV] and property (2) in previous section. This implies that our MTR rendering algorithm does not cost any additional reordering computation.

Fig.4 depicts the whole MTR rendering process of the simple mesh in Fig.1, and its face list F is shown in Table 2. We can see that the required buffer size $b=9$ in this example.

IV. Experiment results

We test our joint compression and MTR rendering algorithm on several typical meshes with different genus. Table.1 shows their experiment results, where b is the required buffer size, $C=b/\sqrt{|V|}$. We can see that the compression ratios of connectivity are all less than 2.5 bits/triangle, which is comparable to some reported compression schemes. From our experiments, we found a few factors that affect the compression ratio: (1) initial vertex of traversal: the compression efficiency can be improved more or less if a proper initial vertex is chosen. Unfortunately, we do not have a general guidance for the choice of initial vertex. In current implementation of our algorithm, it has been chosen arbitrarily; (2) regularity of the mesh: like some other compression methods, our compression performance generally depends on the regularity of the mesh, i.e., how uniform is the valence of vertices within the mesh. The more regular the mesh, the fewer left faces will be, and consequently the higher compression ratio can be achieved.

Table 1. Experiment results of joint connectivity compression and MTR rendering algorithm.

3D mesh	$\|V\|$	$\|F\|$	Bits/tri	b	C
Ronny	2486	4968	2.165	100	2.006
Bunny	7124	13784	2.154	164	1.943
Venus	2838	5668	2.302	84	1.577
Mannequin	2732	5420	2.307	117	2.238
Horse	11135	22258	2.048	120	1.137
Crocodile	17332	34404	2.479	245	1.861

As shown in Table 1, the required buffer size b of our MTR rendering is reasonably small. Surprisingly, the constant C for all tested meshes are all around 2, which is much smaller than the upper bound 12.72 that Bar-Yehuda and Gotsman's MTR algorithm [9] achieves in worst case. Instead of using a controllable mesh buffer in [9], the buffer used by our MTR algorithm is a FIFO buffer. Although it is hard to give an upper bound of b for the proposed MTR rendering, through our experimental observation, our MTR algorithm works well in practice. From a theoretical point of view, it would be useful to develop precise mathematical models for "practical" meshes that can be used to understand under what situations would our proposed MTR rendering can be guaranteed to perform well. We are currently working toward this goal [13].

V. CONCLUSIONS

The complexity of 3D models is growing rapidly due to the improved design and model acquisition tools, the widespread acceptance of the technology, and the need for higher accuracy. Over recent years, the delivery of 3D models over the network gain popularity. In interactive modeling, mesh reconstructing has to be done in real time. Therefore, it is urgent to develop bit-efficient formats of 3D meshes for the purpose of both efficient compression and fast rendering.

The proposed algorithm in this paper provides an effective tool for compressing the connectivity of triangle meshes with arbitrary topology, as well as efficient rendering. In order to hide connectivity information of the mesh in the vertex data, we traverse and reorder the vertices in adjacency-list manner, and generate the code bits at the same time. The compression performance is comparable to some of the reported methods. Based on the reorganized representation of the mesh, we present an efficient scheme for rendering the mesh in minimum time. The experiment results show that only a relative small buffer is required by our MTR rendering. Due to the FIFO characteristics of the buffer and the well predictable order of vertices in mesh buffer, our MTR algorithm does not need to search the buffer so as to access the desired vertex data, which speeds up the rendering procedure. Furthermore, the proposed MTR rendering is achieved without any overhead computation. The current version of the proposed algorithm is based on meshes that only contain triangular faces. However, it may be easily generalized to arbitrary polygon meshes.

ACKNOWLEDGEMENTS

This work was supported in part by a postdoctoral research fellowship by the School of Science of Rensselaer Polytechnic Institute. The authors would like to thank Thomas P.-Y. Yu of Department of Mathematical Sciences of RPI for his helpful comments and discussions.

REFERENCES

[1] Hoppe H., Progressive meshes, *SIGGRAPH '96 Proceedings*, pp. 99-180, 1996.

[2] Taubin G. and Rossignac J., Geometric compression through topological surgery, *ACM Trans. on Graphics*, Vol. 17, No. 2, pp. 84-115, Apr. 1998.

[3] Touma C. and Gotsman C., Triangle mesh compression, *Graphics Interface 98 Proceedings*, pp. 26-34, 1998.

[4] Gumhold S. and Strasser W., Real time compression of triangle mesh connectivity, *SIGGRAPH 98 Proceedings*, pp. 133-140, 1998.

[5] Parajola R. and Rossignac J., Compressed progressive meshes, *Technical Report TR-99-05, GVU*. Georgia Tech. 1999.

[6] Deering M., Geometric compression, *SIGGRAPH 95 Proceedings*, pp. 13-20, 1995.

[7] Chow M., Optimized geometry compression for real-time rendering. *IEEE Visualization '97 Proceedings*, pp. 347-354.

[8] Evans F., Skiena S. and Varshney A., Optimizing triangle strips for fast rendering. *IEEE Visualization '96 Proceedings*, pp. 319-326.

[9] Bar-Yehuda R. and Gotsman C., Time/space tradeoffs for polygon mesh rendering, *ACM Trans. on Graphics*, Vol. 15, No. 2, pp. 141-152, Apr. 1996.

[10] Carey R., Bell G., and Martin C., *The virtual reality modeling language ISO/IEC DIS 14772-1*, Apr. 1997, http://www.vrml.org/Specifications.VRML97/DIS.

[11] Thomas H. C., Charles E. L., and Ronald L. R., Introduction to algorithms, *MIT press*, 1990.
[12] Neider J., Davis T. and Woo M. *OpenGL Programming Guide*. Addison-Wesley, 1993.
[13] Lin, G. and Yu T. A non-recursive algorithm for minimum-time rendering of meshes with arbitrary genus. *ACM Trans. on Graphics*, submitted, 2001.

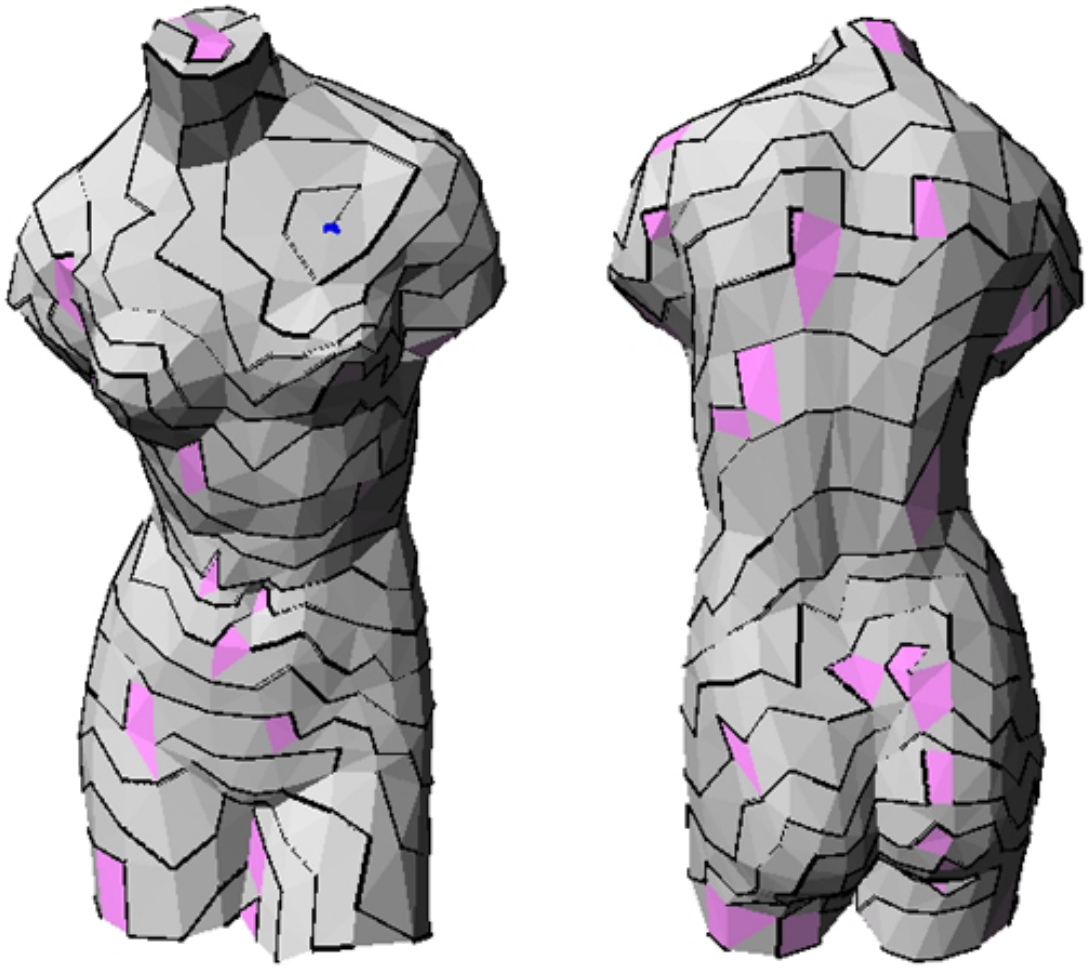


Fig.3. The above figures depict the order of vertex traversal generated by our algorithms (black solid lines). The blue dot is the initial vertex r_1. Notice that while the order of traversal does not necessarily form a path on the graph of the mesh, a large proportion (99.7% in this example) of adjacent vertices $v_i v_{i+1}$ indeed form edges of underlying graph.

Decoded faces	Membership
$f_1 \rightarrow$ (0 1 2)	F_1
$f_2 \rightarrow$ (0 2 3)	F_1
$f_3 \rightarrow$ (0 3 4)	F_1
$f_4 \rightarrow$ (0 4 5)	F_1
$f_5 \rightarrow$ (0 5 6)	F_1
$f_6 \rightarrow$ (0 6 1)	F_1
$f_7 \rightarrow$ (1 6 7)	F_2
$f_8 \rightarrow$ (1 7 8)	F_1
$f_9 \rightarrow$ (1 8 2)	F_1
$f_{10} \rightarrow$ (2 8 9)	F_2
$f_{11} \rightarrow$ (2 9 3)	F_1
$f_{12} \rightarrow$ (3 9 4)	F_2
$f_{13} \rightarrow$ (4 9 10)	F_2
$f_{14} \rightarrow$ (4 10 11)	F_1
$f_{15} \rightarrow$ (4 11 12)	F_1
$f_{16} \rightarrow$ (4 12 5)	F_1
$f_{17} \rightarrow$ (5 7 6)	F_L

Table 2. Decoded faces and their corresponding membership of the simple mesh in Fig.1.

From above table, we can notice that the decoded mesh have the following properties: (1) All faces are reorganized in an increasing order with regard to their first vertex indices; (2) In each face, the first index is the smallest among three indices of its bounding vertices. These properties facilitate MTR rendering of the mesh.

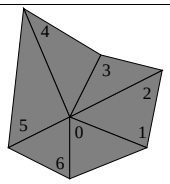
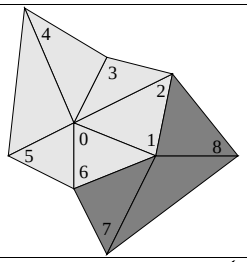
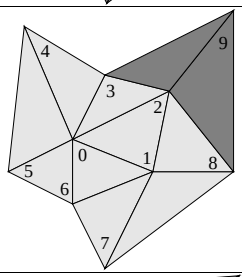
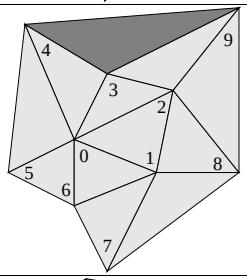
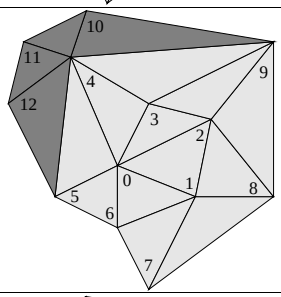
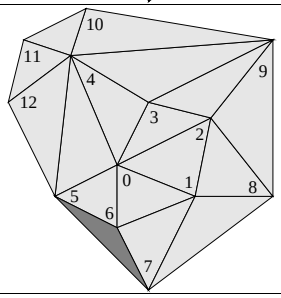
V_b: **0, 1, 2, 3, 4, 5, 6** $v_{focus} = 0$ $F'(v_{focus})$: (0 1 2), (0 2 3), (0 3 4), (0 4 5), (0 5 6)	
V_b: **1, 2, 3, 4, 5, 6, 7, 8** $v_{focus} = 1$ $F'(v_{focus})$: (1 6 7), (1 7 8), (1 8 2)	
V_b: **2, 3, 4, 5, 6, 7, 8, 9** $v_{focus} = 2$ $F'(v_{focus})$: (2 8 9), (2 9 3)	
V_b: **3, 4, 5, 6, 7, 8, 9** $v_{focus} = 3$ $F'(v_{focus})$: (3 9 4)	
V_b: **4, 5, 6, 7, 8, 9, 10, 11, 12** $v_{focus} = 4$ $F'(v_{focus})$: (4 9 10), (4 10 11), (4 11 12), (4 12 5)	
V_b: **5, 6, 7, 8, 9, 10, 11, 12** $v_{focus} = 5$ $F'(v_{focus})$: (5 7 6)	

Fig.4. MTR rendering of the example mesh.