A Breadth-First Approach To Efficient Mesh Traversal

Tulika Mitra Tzi-cker Chiueh

Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{mitra,chiueh}@cs.sunysb.edu

Abstract

Complex 3D polygonal models are typically represented as triangular meshes, especially when they are generated procedurally, or created from volumetric data sets through surface extraction. Existing 3D rendering hardware, on the other hand, processes one triangle at a time. Therefore triangle meshes need to be converted to individual triangles when they are fed to the graphics pipeline. The design goal of such conversion algorithms is to minimize the number of vertices that are sent redundantly to the rendering pipeline. This paper proposes a breadth-first approach to traverse triangle meshes that reduces vertex redundancy to very close to the theoretical minimum. With the proposed scheme, no triangle vertices need to be specified multiple times, barring exceptional cases. In addition, owing to a prefetching technique, the on-chip storage requirement for effective mesh traversal remains small and largely constant regardless of the mesh size. Our experimental results show that assuming a 64-vertex buffer, the redundant transformation overhead associated with the proposed approach is between 1.00% and 7.33%, for a set of 8 triangle meshes whose size ranges from 2,992 to 40,000 triangles.

Keywords: triangular mesh, 3D polygonal rendering, breadth-first traversal, prefetching.

1 Introduction

Polygonal graphics rendering systems consist of a geometric transformation engine and a rasterization engine. Although high-end graphics systems such as SGI's dedicate special hardware to both transformation and rasterization, most low-end PC-class systems only support rasterization in hardware. As a result, the geometric transformation stage tends to become the bottleneck of PC graphics pipelines. Our measurement shows that a 233-MHz Pentium-II processor can transform about 200K triangles per second, while a Voodoo 3DFX graphics card can rasterize up to 350K triangles per second even when advanced features such as texture mapping, fogging, and anti-aliasing are all turned on. The motivation of this work is to reduce the geometric transformation load by eliminating redundant vertex processing during traversal of the 3D model database.

3D polygon models are typically represented as triangle meshes for storage efficiency, especially those that are generated automatically from programs. However, existing graphics systems render 3D objects on a triangle-by-triangle basis. Therefore, conversion from triangle meshes to individual triangles is necessary. A naive conversion algorithm is to send each triangle in the mesh to the graphics system independently of each other. If a vertex participates in X triangles in the mesh, it will be sent X times. This redundancy not only increases the input traffic of the graphics subsystem, but also imposes additional workload on the geometric transformation subsystem because each incoming vertex is processed and transformed independently.

OpenGL, the de facto 3D application programming interface maintains a 2-entry stack for vertices that are already sent to the graphics system. That is, a new triangle can be constructed by specifying a new vertex together with two vertices already in the stack. Vertices of a new triangle that are not currently in the stack need to be specified explicitly and thus processed. After a triangle is formed, the new vertex of the triangle is then pushed to the stack, displacing the oldest stack entry at that time. In addition, GL provides a *swap* primitive to exchange the positions of the stack entries. This vertex storage model is optimized for triangular strips, where a linear sequence of triangles can be specified at the cost of one vertex per triangle asymptotically. For general triangular meshes, such storage models are inefficient in that a significant percentage of vertices need to be specified multiple times. It is possible to reduce the percentage of redundant vertices by employing a more general storage model than stacks to hold vertices that are already sent to the graphics system. For example, a large register set should offer more flexibility in vertex reuse, thus reducing the need of specifying vertices redundantly and the associated transformation overhead.

Intuitively there is a space/time tradeoff between the register set size and the degree of vertex reusing. The approach proposed in this work attempts to push this space/time tradeoff to one extreme by minimizing the redundancy at the expense of storage cost. More specifically, the algorithm traverses a triangular mesh in the breadth-first order.

Each triangle vertex in the mesh is explicitly represented exactly once in most cases. The breadth-first traversal scheme has the desirable property that the vertex access pattern is predictable. Consequently, only a small window of the vertex buffer needs to be present on chip, and perfect prefetching of the next window is possible. The on-chip storage requirement for efficient mesh traversal is thus relatively small and is independent of the mesh size.

The rest of this paper is organized as follows. Section 2 reviews previously proposed mesh traversal methods and related efforts in reducing the overhead of geometric transformation in 3D rendering. Section 3 describes the proposed mesh traversal method and its software and hardware implementation strategies. Section 4 presents the results and analysis of a performance study of the proposed approach. Section 5 concludes this paper with a summary of the research results, and an outline of current work.

2 Related Work

OpenGL [8] supports *triangular strips*, and the SGI hardware dedicates three registers for vertex storage to improve its efficiency. GL [9] in addition provides a *swap* command to support *generalized triangle strips*. Akeley et. al. [1] implemented a heuristic to convert triangular meshes to triangle strips. Evans et. al. [6] improved upon this basic idea by proposing a global patchification scheme to detect large strips of quads and to dynamically triangulate partially triangulated 3D models. Speckmann et. al. [10] proposed a faster scheme to create triangle strips by traversing a special spanning tree of the dual graph of the input mesh in a modified depth-first fashion.

Deering [4] proposed an extension of the existing hardware architecture to store vertices in a finite stack, so that an already visited vertex need not be respecified if it exists in the stack. This is known as *generalized triangle mesh* and it can potentially encode a mesh by specifying each vertex exactly once. However, Bar-Yehuda et. al. [12] showed that a stack of size at least $1.649\sqrt{n}$ is required to render a triangle mesh on n vertices by sending each vertex exactly once. This suggests that a prohibitively large stack size would be required for big models. Taubin et. al. [11] proposed an efficient method to compress the connectivity information by using just 2 bits per triangle. However, they assumed random access to all vertex co-ordinates during decompression which would require a large on-chip cache memory. Following this work, Chow [3] proposed various algorithms to construct generalized triangular meshes that would require average $0.67t$ vertices for t triangles using only 16 buffer entries. The theoretical minimum is $0.5t$ vertices since n vertices can form at most $2n$ triangles.

Denny et. al. [5] presented a completely different approach of encoding a straight-edge triangulation as a permutation of its point set, and decoding it efficiently. But this algorithm is more of theoretical interest as real models might not have regular triangulation.

This work differs from all the work done so far in that the proposed scheme traversed the mesh in the breadth-first order, and as a result each traversed triangle requires at most one explicitly represented vertex. In addition, effective prefetching significantly reduces the amount of on-chip storage required, and is possible because of predictable access patterns. Both [3] and [6] chose to use small buffers due to hardware limitations. Francine et. al. [6] showed that large buffers only gave minimum improvement in vertex reuse. We show that it is possible to reuse vertices effectively using a relatively large buffer provided that the accesses to the memory is sequential. The sequential access pattern together with the alternate encoding scheme makes it possible to specify $0.52t$ vertices for a triangular mesh of t triangles, which is very close to the theoretical minimum.

3 Efficient Mesh Traversal

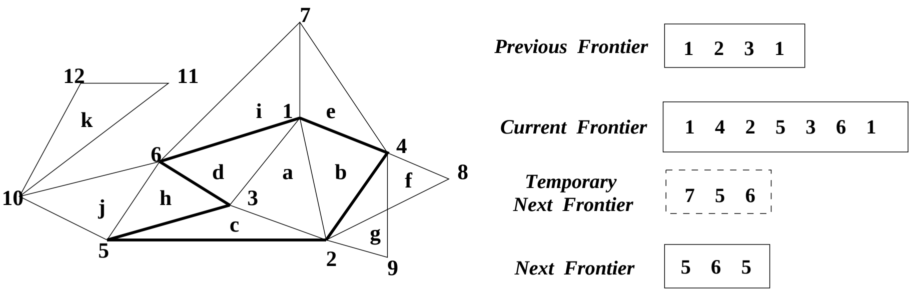
3.1 Breadth-First-Traversal

The goal of mesh traversal is to enumerate all the triangles in a given triangle mesh, and to send them to the graphics pipeline. By carefully traversing through the mesh, one can reuse the vertices previously sent into the graphics subsystem to enumerate subsequent triangles, thus avoiding the redundant vertex problem. The efficiency of mesh traversal therefore is strongly dependent on the storage interface that the graphics subsystem exposes to the mesh traversal program.

In earlier works, the mesh traversal program assumed a stack-like storage interface so that each new triangle is constructed using a new vertex, which is explicitly represented, and the last two vertices sent into the graphics pipeline. These mesh traversal schemes visit the vertices in the mesh in the depth-first order, since it works well with the stack storage model. However, depth-first traversal invariably leads to significant redundant vertices, because the periphery vertices of the strips created by depth-first traversal can not be easily reused and thus need to be specified redundantly. The traversal algorithm we proposed, called *BFT* (Breadth-First Traversal) traverses the triangle mesh in the breadth-first order. The performance goal is to explicitly specify only one new vertex per enumerated triangle and no vertex needs to be specified or processed more than once even when it participates in multiple triangles.

Given a triangle mesh, BFT first picks a starting triangle, whose edges form the *current frontier*. Formally a *frontier* is an ordered sequence of vertices. Each frontier forms an imaginary polygon. At each iteration, BFT essentially enumerates all the triangles that have not been accessed previously and that share an edge with the imaginary polygon corresponding to the *current frontier*. More concretely, BFT visits each edge of the *current frontier*, and incrementally constructs the *next frontier*. For each edge, BFT attempts to pair it with a third vertex to represent a triangle in the mesh. In certain cases, no triangle can be formed from the current edge. Such edges are called *null edges*.

The third vertex used in enumerating a triangle could be explicitly represented in terms of its coordinates, or some vertex that appeared previously and thus could be represented implicitly as a pointer into the *current frontier* or *next frontier* buffers. These pointers are specified as offsets with respect to the *active cursors* of these two buffers. For the *current frontier* buffer, the *active cursor* is the first vertex of the edge that BFT visits currently. For the *next frontier* buffer, the *active cursor* is the last vertex added to the *next frontier* buffer. In general, the *next frontier* con-



Command Sequence

1. no-copy, non-null, new vertex 7, temporary-copy, no-next tri

4. no-copy, non-null, new vertex 8, no-copy, next-triangle, new vertex 9, no-copy,no next tri

2. no-copy, null

5. permanent copy, non-null, current frontier pointer 2, permanent-copy,no-next tri

3. no-copy, null

6. permanent-copy, non-null, next frontier pointer 2, no-next tri

Figure 1: *An example mesh that illustrates how BFT enumerates triangles in the mesh by visiting each edge of the current frontier and constructing the next frontier. The current frontier in this case is the vertex sequence (*1, 4 , 2 , 5 ,3 ,6*), shown in bold. Vertex 1 is duplicated to maintain continuity. The triangles enumerated at this level are e, f, g, h, and i. Triangles e, f, and g introduce new vertices whereas h refers to an entry in the current frontier, and i refers to an entry in the next frontier. Since Vertex 7 is only needed at this level, it is temporarily added to the next frontier but is never written back to the next frontier buffer. This is a case of* no copy*. Triangle k is a corner triangle, as it can not be covered by BFT. The command sequence and the status of the current and next frontier buffers are also shown.*

sists of vertices from the *current frontier* and new vertices that are represented explicitly. A vertex is added to the *next frontier* if and only if there is at least one un-visited triangle that uses this vertex.

After visiting the last edge of the *current frontier*, BFT starts the next level by converting the *next frontier* into the *current frontier*, and initializing the *next frontier* to a null sequence. To maintain continuity, the first vertex of the first edge of the *current frontier* is duplicated as the second vertex of its last edge. BFT continues this traversal, level by level, until it reaches the level for which the *next frontier* remains null when BFT hits the end of that level. There is no guarantee that the BFT algorithm will cover all the triangles in a given triangle mesh. Those triangles that can not be enumerated by BFT are called *corner triangles* and need to be explicitly represented with its three vertices. However, corner triangles are very rare in triangle mesh of a real object.

For an input mesh, BFT first pre-processes it to find out the vertex visiting order and the *current* and *next frontiers* at each level, and encodes the mesh into a command sequence appropriately. At rendering time, application software sends the command sequence to the the graphics engine, which constructs triangles and maintains the *current* and *next frontier* buffers according to the instruction encoded in the command sequence. Figure 1 shows the instructions sent to the graphics engine when the current buffer consists of vertices 1, 4, 2, 5, 3, 6. Each command in the command sequence corresponds to an edge in the *current frontier* and includes the following:

1. status of the active cursor vertex
   (a) *permanent copy* - copy this vertex to the next frontier permanently
   (b) *temporary copy* - copy this vertex to the next frontier temporarily
   (c) *no copy* - never copy this vertex to the next frontier
2. triangle(s) sharing the edge formed by the current vertex and its adjacent vertex
   (a) *null edge* - no triangle associated with this edge
   (b) *non-null edge* - triangle(s) associated with this edge
      i. the third vertex of the first triangle
         A. *current frontier pointer <pointer>*
         B. *next frontier pointer <pointer>*
         C. *new vertex <coordinates> <normals>*
      ii. status of the third vertex
         A. *permanent copy* - copy this vertex to the next frontier permanently
         B. *temporary copy* - copy this vertex to the next frontier temporarily
         C. *no copy* - never copy this vertex to the next frontier
      iii. *next triangle or no next triangle* depending on whether there is any more triangle associated with this edge

If no future triangle is going to reference a vertex, the vertex need not be copied to the *next frontier*. This corresponds to the *no copy* case. If some future triangles at the current level but not following levels need to reference a vertex, the vertex is temporarily copied to the *next frontier* in the sense that it is removed when BFT advances to the next level. Finally, if a vertex is going to be referenced by triangles at future levels, it is permanently copied to the *next frontier* and stays there as the *next frontier* is converted into the *current frontier* at the next level. This requires 1 bit for the *no-copy* case and 2 bits for either *permanent copy* or *temporary copy*.

Next, a 1-bit flag specifies whether there is any triangle to be constructed from the current edge. If the current edge is associated with at least one triangle, then for each triangle, a third vertex needs to be specified. Whether the third vertex is explicitly represented, or implicitly represented as pointers to the current or next frontiers, is distinguished by a 2-bit flag. When a vertex explicitly represented, it takes 24 bytes to specify its coordinates and normal vectors. When it is implicitly represented as a pointer to the current/next frontier, 5 bits are required because we assume the active windows of both frontier buffers have 32 entries each. In the latter case, a vertex that participates in multiple triangles only needs to be explicitly represented once, which leads to reduction in input traffic to the graphics system as well as in redundant vertex transformation. The application also needs to tell the graphics engine, by using 1 or 2 bits, as to whether the third vertex should be copied to the *future frontier* and how. Note that each edge in the *next frontier* could be associated with 0, 1 or multiple triangles. Hence, one bit is required to specify whether there are any more triangles that are associated with the current edge.

BFT achieves the goal of specifying only one vertex per triangle in most cases. Vertices that are visited previously are referenced as indices into the *current frontier* and *next frontier* buffers. Provided that we have infinite space for these two buffers, the only vertices that need to be specified redundantly are the vertices of the *corner triangles*. Another overhead of BFT is the bits required to specify *null edges*, which is specific to BFT because BFT has to visit every edge in the current frontier. The original version of BFT did not distinguish between *permanent copy* and *temporary copy* when putting a vertex into the *next frontier*. As a result, about 50% of the edges visited were *null edges*. With *temporary copy*, the percentage drops to 20%, because every vertex in the *current frontier* is guaranteed to contribute to at least one triangle in the current or following levels.

3.2 Prefetching of Active Frontier Window

The size of the current and next frontier buffer grows in proportion to the maximum width of the breadth-first traversal tree. For large triangle meshes, the space requirement of these two frontier buffers for supporting BFT could be too large to prevent them from being maintained on chip, thus leading to longer execution time than expected due to off-chip memory access overheads. Fortunately, because BFT visits the vertices in a highly predictable way, perfect prefetching for the frontier buffers is possible.

The key observation is that if the third vertex that pairs with the current edge to form a triangle is not represented explicitly, it is in most cases a vertex that falls within a certain distance from the *active cursor* of the current or the next frontier buffer. As a result, such vertices are represented as offsets from the active cursor into the current or next frontier buffer. For the current frontier, the third vertex could only appear after the active cursor. For the next frontier, the third vertex could only appear before the active cursor. Because of this access pattern, the only portion of the current/next frontier buffers that need to be present is a small percentage of the total frontier buffer size. Moreover, the fact that the active cursors of the frontier buffers are known at run time, implies that its surrounding windows of a fixed size could be prefetched perfectly.

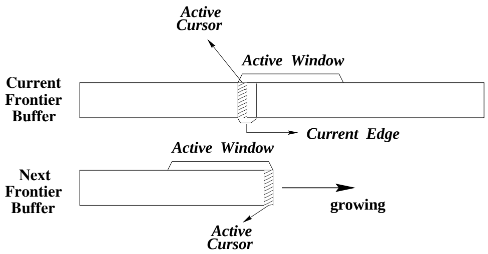


Figure 2: *While BFT traverses through the current frontier to enumerate triangles, only the active windows of the current and next frontier buffers need to be kept on chip. Every time BFT advances to the next edge in the current frontier, the active window of the current frontier also advances by one vertex entry, and the active window of the next frontier can advance by 0, 1, or more entries.*

Whereas the entire current and next frontier buffers are assumed to reside in off-chip memory, the active windows of the current/next frontier buffer are each organized as an on-chip circular FIFO queue, as shown in Figure 2. Every time the active cursor of the current frontier moves forward, the vertex following the active window is brought in from the current frontier buffer in off-chip memory and replaces the oldest entry in the window. When a vertex is added to the next frontier, it replaces the oldest entry in the next frontier's active window. If the replaced entry is marked as *permanent copy*, it has to be written back to the off-chip next frontier buffer first. For *temporary copy* entries, the write back is not necessary.

After visiting an edge in the current frontier, BFT brings in a vertex from the current frontier buffer, and writes back one or multiple vertices to the next frontier buffer. Every time a triangle is constructed, the third vertex needs to be transformed first if it is explicitly represented, before it could be added to the next frontier buffer. The vertices in the current frontier buffer are already transformed by construction. Then the resulting transformed triangle goes through the rasterization stage. The time required for processing triangles generated on an edge visit is expected to be longer than the time for vertex read/write accesses to the off-chip current/next frontier buffers. For example, the time to transform and rasterize a triangle is on the order of 1 μsec, whereas reading and writing a vertex (24 bytes) should take less than 500 nsec. Therefore, the read/write delay to maintain active frontier windows is completely masked. The only scenario in which this is not true is when the current edge is a null edge. Fortunately our measurements in the next section show that null edges rarely occur in bursts and therefore can not result in serious backlogs.

Maintaining only frontier windows rather than the entire frontier buffers on chip incurs a performance overhead. That is, when the third vertex falls out of the frontier window, it can not be represented implicitly via a pointer. Such vertices are called *lost vertices* and need to be explicitly represented, thus incurring redundancy.

4 Performance Evaluation

4.1 Datasets

We choose eight 3D mesh datasets to evaluate the performance of BFT. Six of those datasets are from Avalon 3D

Dataset	No. of Vertices	No. of Triangles
plane	1508	2992
skyscraper	2022	3692
triceratops	2832	5660
power lines	4091	8966
honda	7106	13594
dodge	8477	16646
figure	9999	20000
skull	19999	40000

Table 1: *The characteristics of the eight triangle meshes used in this study.*

models, and two (figure and skull) are scientific datasets. These datasets vary widely in the numbers of vertices and triangles. Table 1 shows the characteristics of the datasets. Our preprocessing algorithm performs breadth first traversal of the triangle mesh to find out the traversal order and the current and next frontier for each level. The pre-processing stage takes only one pass through the dataset.

4.2 Results and Analysis

Since the major goal of this work is to minimize the overhead due to redundant vertex transformation, Table 2 shows the additional transformation cost for the eight test meshes under BFT. The minimum number of vertex transformations required is the same as the number of vertices. BFT comes very close to this minimum, with the additional overhead ranging from 1% to 7.33%. There are two sources of redundant vertices: one is due to corner triangles, which are triangles that can not be covered by BFT, and the other is due to lost vertices, which are vertices that can not be reused because they fall out of the active window of the current/next frontiers. Each corner triangle costs three redundant vertices.

The other consideration for efficient triangle mesh traversal is to reduce the input traffic volume to the 3D graphics pipeline. Table 3 shows the number of bytes required to traverse each test mesh. The **Vertex Cost** column represents the cost to transfer the coordinate information for the vertices in a mesh. This is equal to the number of vertices multiplied by 24. Each corner triangle costs an additional 24x3 = 72 bytes. Each lost vertex costs an additional 24 bytes. The **Pointer Cost** column represents the cost due to index pointers when the third vertex is referenced from the active windows of the current/next frontier buffers. If the size of each active window is S, the number of bits per pointer is $\log_2 S$. The **Command Cost** is the cost spent in instructing the graphics subsystem how to enumerate triangles and construct the next frontier while traversing the current frontier. Each null edge costs a bit to indicate that it is not associated with any triangle. The null edge cost is specific to BFT. Since the output of a mesh traversal scheme is to provide the graphics subsystem a sequence of triangles, the best way to evaluate a traversal scheme's space efficiency is to measure the number of bytes required to represent which three vertices constitute each triangle in the mesh. The last column in Table 3 provides exactly this measure for BFT. The average per-triangle overhead for specifying the components of each triangle is between 1.03 to 1.83 bytes.

Dataset	Max Memory Requirement (Bytes)	Max Frontier Size (no of entries)
plane	3288	128
skyscraper	3120	121
triceratops	5400	214
power lines	2664	102
honda	1848	71
dodge	4416	177
figure	5448	218
skull	16140	671

Table 4: *The off-chip storage requirement for the test meshes under BFT, which is proportional to the width of the breadth-first tree.*

Dataset	1	2	3	4	5	> 5
plane	78.41	13.80	6.72	1.08	0.00	0.00
skyscraper	78.67	15.37	3.94	1.21	0.00	0.00
triceratops	84.68	10.12	4.05	0.81	0.00	0.00
power lines	34.44	19.44	8.68	7.67	4.31	25.46
honda	84.88	13.75	1.15	0.23	0.00	0.00
dodge	91.53	6.66	0.61	0.07	0.00	1.03
figure	87.60	10.82	1.22	0.15	0.09	0.11
skull	92.89	6.21	0.61	0.24	0.05	0.00

Table 5: *The distribution of the length of null edge bursts. Each column corresponds to one length. Each table entry represents the percentage of null edge bursts in a data set with a certain length.*

A major concern with breadth-first mesh traversal schemes is the size of the frontier buffers, although the frontier buffers are assumed to be reside in off-chip memory. Table 4 shows the maximum frontier size for the test meshes in terms of the number of vertices and the actual number of bytes. Because the next frontier buffer is dynamically growing while the useful part of the current frontier buffer is dynamically shrinking, the number of bytes required for a mesh reported in Table 4 is the maximum of the sum of the sizes of the next frontier buffer and the useful part of the current frontier buffer. The sizes of the frontier buffers do not appear to be proportional to the size of the triangle mesh.

The key claim of the proposed BFT scheme is that at any point in time, only a small window of the current/next frontier buffers is needed to allow vertex reuse. As a result, the on-chip window storage requirement is small and is independent of the input mesh size. Figure 3 demonstrates that this is indeed the case by showing the third vertex hit ratio versus the active window size for the current frontier buffer. If the window size is N, and the third vertex of a triangle to be enumerated is located within N entry from the active cursor of the current frontier, such a third vertex reference is considered a hit. The hit ratio is calculated by using the total number of vertex references that fall in the current frontier buffer as the denominator.

At a window size of 128 entries, close to 100% hit ratios are observed for all 8 test meshes. 64-entry window also gives very high hit ratios for all test meshes, better than 97%. The vertex reference hit ratio for the next frontier

Dataset	Minimum No. of Transformations	No. of Corner Triangles	No. of Lost Vertices	Amount of Redundancy
plane	1508	0	71	71 (4.71%)
skyscraper	2022	0	30	30 (1.48%)
triceratops	2832	0	106	106 (3.74%)
power lines	4091	56	132	300 (7.33%)
honda	7106	0	118	118 (1.66%)
dodge	8477	0	393	393 (4.63%)
figure	9999	0	565	565 (5.65%)
skull	19999	0	199	199 (1.00%)

Table 2: *The amount of redundant transformation overhead for the test meshes under BFT. The percentages are calculated with respect to the minimum number of transformations required.*

Dataset	Vertex Cost	Corner Triangles	Lost Vertices	Pointer Cost	Command Cost	Null Edges	Per-Triangle Overhead
plane	36192	0	1704	891	2395	140	1.71
skyscraper	48528	0	720	1135	2815	119	1.30
triceratops	67968	0	2544	1703	1415	187	1.03
power lines	98184	4032	300	3036	8257	755	1.83
honda	170544	0	2832	4016	10738	610	1.34
dodge	203448	0	9432	4996	12830	633	1.68
figure	239976	0	13560	5497	15341	660	1.75
skull	479976	0	4776	12375	30276	1241	1.22

Table 3: *The bandwidth overhead in numbers of bytes for the test meshes under BFT. The last column shows the per-triangle space overhead.*

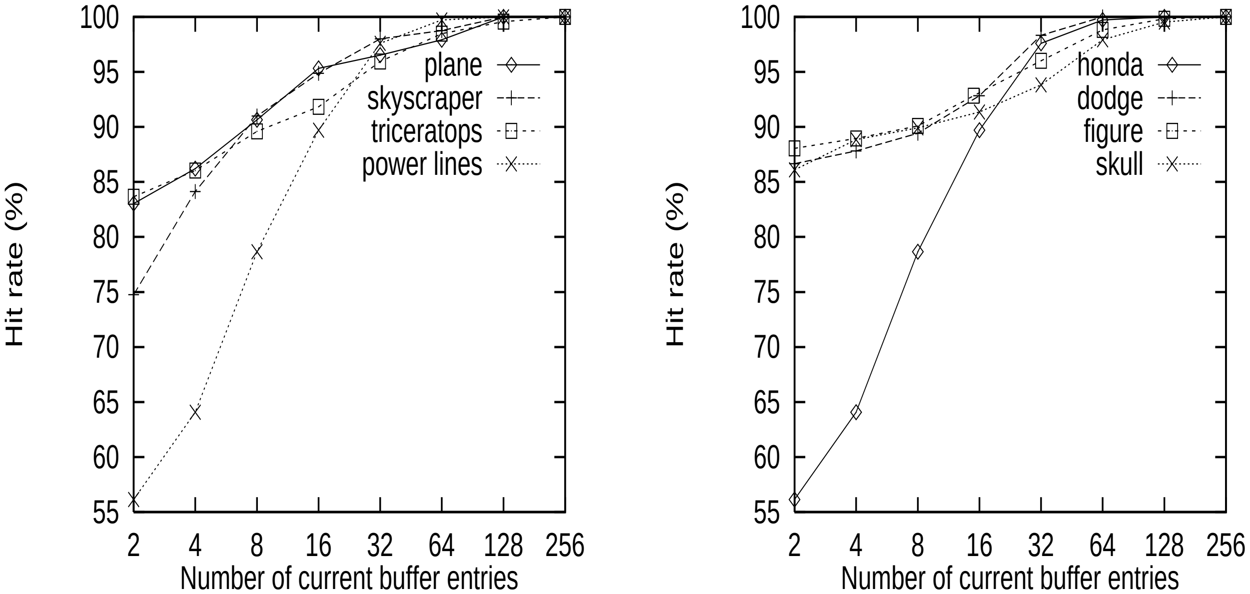


Figure 3: *The vertex reference hit ratio versus the size of the active window for the current frontier buffer, for the eight test meshes. Each hit ratio measurement for a given window size, S, represents the percentage of all vertex references to the current frontier buffer that are within S entries from its active cursor.*

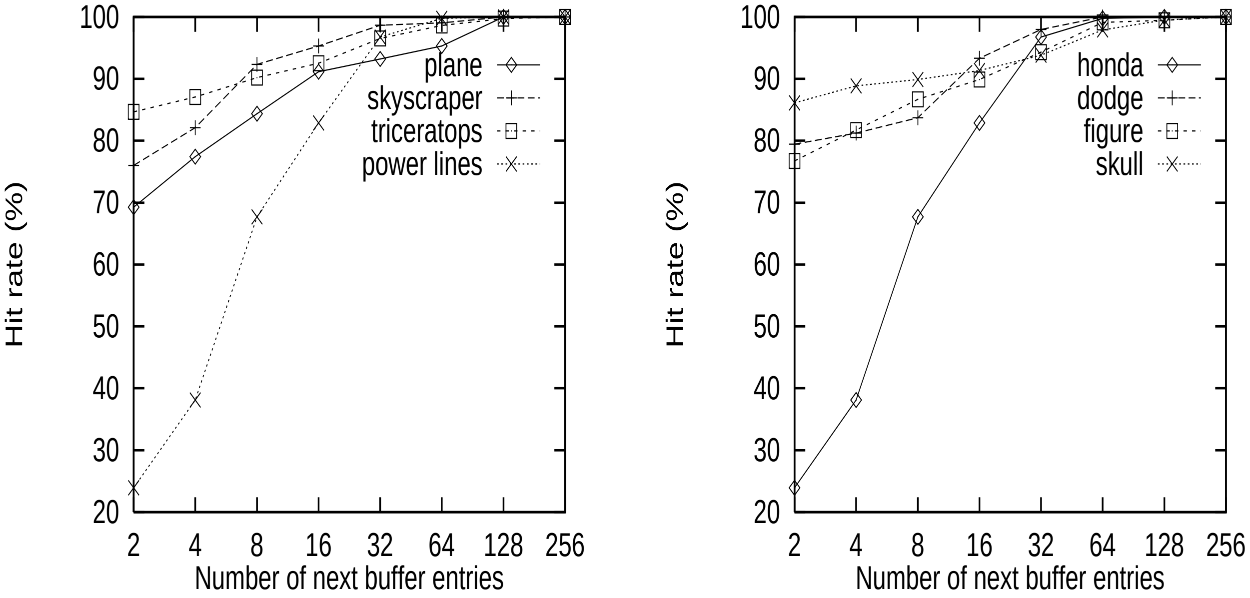


Figure 4: *The vertex reference hit ratio versus the size of the active window for the next frontier buffer, for the eight test meshes. Each hit ratio measurement for a given window size, S, represents the percentage of all vertex references to the next frontier buffer that are within S entries from its active cursor.*

buffer is similarly defined and exhibits almost identical behaviors, as shown in Figure 4. These curves demonstrate that there is strong spatial locality among the three vertices that constitute a triangle, and the degree of this locality, as reflected by the window size required to attain a fixed hit ratio, is independent of the mesh size. These measurements show that a total of 128 entries – 64 for the current frontier and 64 for the next frontier – is sufficient to attain high vertex reuse.

Small active frontier window size is only possible if prefetching is effectual. As explained earlier, the only scenario in which the data access delay associated with active window management can not be masked is when BFT encounters a consecutive burst of null edges. Fortunately, most of the null edge burst is of length 1, as shown in Table 5. That is, null edges occur in isolation most of the time. This proves that the off-chip memory accesses necessary for maintaining the active windows can effectively overlap with triangle processing.

The performance of BFT, in terms of amounts of redundant transformation, depends on the choice of the triangle that starts the traversal. Table 6 shows the comparison of amount of redundant transformation among four possible choices of the starting triangle. The **Center** approach chooses the triangle in the center of the mesh. The **Maximum** and **Minimum** approaches choose the triangle whose center has the largest and smallest magnitude. The **Maximum Adjacency** approach chooses the triangle with maximum connectivity. There is no clear winner among the four choices, although the performance difference among them could be significant. More work on how the choice of the starting triangle affects BFT's performance is needed.

Finally, Table 7 shows the pre-processing time required to convert a triangle mesh into the BFT command sequence. These measurements are taken from a PentiumPro 200-MHz machine under FreeBSD Unix. This measurement is of interest because in certain cases on-line conversion is required, e.g., displaying 3D models downloaded from the network. In these cases, the pre-processing time should not out-weigh the saving from reducing redundant vertex transformation.

5 Conclusion

The original motivation of this work was to reduce the geometric transformation overhead in 3D polygon rendering. This is particularly important when geometric transformation is performed by software running on general-purpose CPU, as in current PCs. Because an increasing percentage of polygon models in 3D applications are represented as triangle meshes, an efficient mesh traversal scheme that minimizes redundant vertex transformation significantly reduces the overall transformation load. This paper describes an efficient mesh traversal scheme in detail and its evaluation. The key insight is that vertex redundancy during mesh traversal can be completely eliminated if there is sufficient buffer space in the graphics pipeline, and as long as the vertex access pattern is predictable, the size of the indispensable portion of the vertex buffer can be kept small and fixed. Based on this insight, the proposed approach is breadth-first rather than depth-first traversal. Our experiment shows that the mesh traversal scheme proposed in this paper reduces on the average the redundant transformation to less than 8% of the transformation work inherent in the 3D datasets. Moreover, through prefetching, the proposed scheme achieves this performance level using only a 64-vertex buffer, regardless of the input mesh size.

This work is performed in the context of a parallel 3D graphics engine project called *Sunder*, which integrates state-of-the-art PC graphics cards with gigabit/sec system area network technology. We are currently implementing the proposed mesh traversal scheme in Mesa, a public-domain OpenGL-compatible polygon rendering tool. In addition, we are exploring several variants of the proposed approach. First, in this work we assume that the 3D dataset is pre-

Dataset	Center	Minimum	Maximum	Maximum Connectivity
plane	72(4.77)	47(3.12)	99(6.56)	71(4.71)
skyscraper	63(3.12)	91(4.50)	59(2.92)	30(1.48)
triceratops	130(4.59)	121(4.27)	141(4.98)	106(3.74)
power lines	298(7.28)	217(5.30)	182(4.45)	300(7.33)
honda	97(1.36)	112(1.58)	63(0.88)	118(1.66)
dodge	414(4.88)	340(4.01)	331(3.90)	393(4.63)
figure	545(5.45)	537(5.37)	499(5.00)	565(5.65)
skull	261(1.30)	330(1.65)	345(1.73)	199(1.00)

Table 6: *The impact of different choices of the starting triangles on the amount of redundant transformation. The numbers in the parenthesis are percentages with respect to the minimum number of transformations.*

Dataset	Preprocessing Time (sec)
plane	0.03
skyscraper	0.10
triceratops	0.05
power lines	0.15
honda	0.18
dodge	0.45
figure	0.23
skull	1.21

Table 7: *The pre-processing time of the BFT scheme to convert a triangle mesh into a command sequence for the test meshes. Measurments are taken from a PentiumPro 200-MHz machine running FreeBSD UNIX.*

parsed, and therefore the "compilation" time is ignored. In practice, 3D data sets could be downloaded from the network in real time such as VRML files. Therefore efficient run-time conversion is essential for the graphics system to employ the proposed scheme in this case. Second, in this work we assume that the entire triangle mesh need to be traversed during rendering. However, in architectures that support viewpoint-directed model traversal [2], this may not be the case. However, combination of viewpoint-directed traversal with efficient triangle enumeration is critical to reduce the overall geometric transformation overhead.

Acknowledgement

This research is supported by an NSF Career Award MIP-9502067, NSF MIP-9710622, NSF IRI-9711635, a contract 95F138600000 from Community Management Staff's Massive Digital Data System Program, as well as fundings from Sandia National Laboratory, Reuters Information Technology Inc., and Computer Associates/Cheyenne Inc.

REFERENCES

[1] K. Akeley, P. Haeberli, and D. Burns. Tomesh.c: C Program on SGI Developer's Toolbox CD, 1990.

[2] Tzi-cker Chiueh. Heresy: A Virtual Image-Space 3D Rasterization Architecture. In Proceedings of ACM SIGGRAPH/Eurographics Graphics Hardware Workshop, August, 1997.

[3] Mike. M. Chow. Optimized Geometry Compression For Real-Time Rendering. In Proceedings of the IEEE Visualization, 1997.

[4] M. Deering. Geometry Compression. In SIGGRAPH 95 Conference Proceedings.

[5] M. O. Denny and C. A. Sohler. Encoding A Triangulation As A Permutation Of Its Point Set. In Proceedings of the Ninth Canadian Conference on Computational Geometry, August 1997. (Electronic proceedings available at http://www.dgp.toronto.edu/cccg/cccg97)

[6] F. Evans, S. Skiena, and A. Varshney. Optimizing Triangle Strips For Fast Rendering. In Proceedings of the IEEE Visualization, 1996.

[7] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification Of Parallel Rendering. IEEE Computer Graphics and Applications, 14(4):23-32, July 1994.

[8] Jackie Neider, Tom Davis, and Mason Woo. OpenGL Programming Guide. Addison-Wesley, June 1995. ISBN 0-201-63274-8.

[9] Silicon Graphics Inc. Graphics Library Programming Guide. 1991.

[10] B. Speckmann and J. Snoeyink. Easy Triangle Strips For TIN Terrain Models. In Proceedings of the Ninth Canadian Conference on Computational Geometry, August 1997. (Electronic proceedings available at http://www.dgp.toronto.edu/cccg/cccg97)

[11] G. Taubin and J. Rossignac. Geometry Compression Through Topological Surgery. IBM RC-20340, 1996. (available at http://www.research.ibm.com/vrml/binary)

[12] R. Bar-Yehuda and C. Gotsman. Time/Space Tradeoffs For Polygon Mesh Rendering. ACM Transaction on Graphics, 15(2):141-152, April 1996.

[13] T.R. Halfhill. Beyond Pentium II [IA-64]. BYTE (International Edition), 22(12):80-86, December 1997.