

Compression-Domain Parallel Rendering

Tulika Mitra
School of Computing
National University of Singapore
Singapore 117543
tulika@comp.nus.edu.sg

Tzi-cker Chiueh
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
chiueh@cs.sunysb.edu

Abstract

Three dimensional triangle mesh is the dominant representation used in parallel rendering of 3D geometric models. However, explosive growth in the complexity of the mesh-based 3D models overwhelms the communication bandwidth of existing parallel rendering systems. An effective solution to this problem is to use a compressed mesh representation. In recent years, researchers have shown a great deal of interest in developing highly efficient mesh compression algorithms. However, using compressed mesh in the parallel rendering architecture to achieve highest end-to-end performance is a largely unexplored area. We have earlier developed an efficient mesh compression/decompression algorithm, called Breadth First Traversal (BFT) [3, 4]. In this work, we design and implement a parallel rendering architecture that can use a BFT mesh representation. The enabling technology is a novel algorithm that can perform a compression-domain subdivision of the BFT mesh for bandwidth-efficient distribution of sub-meshes to parallel processors. Parallel rendering using BFT mesh reduces the communication requirement to about one third of that of uncompressed representation.

1 Introduction

Three-dimensional graphic rendering pipeline converts the geometric representation of a 3D virtual world to a photo-realistic 2D image. The input to the pipeline is a scene consisting of a set of objects that are typically represented as triangle meshes (set of triangles). The 3D graphics pipeline itself consists of two distinct stages: *geometric transformation* and *rasterization* [1]. The geometric transformation stage maps triangles from a 3D coordinate system (object space) to a 2D coordinate system (image space) by performing a series of transformations. The rasterization stage converts transformed triangles into pixels.

The 3D graphics pipeline is computation intensive, but is

quite amenable to parallel implementation. The parallelization strategies can be classified as sort-first, sort-middle, and sort-last depending on where the sorting operation (i.e., distribution of primitives to different processors) is performed [5]. In this paper, we will concentrate on *sort-first architecture* even though the techniques are generic enough to be applied to both sort-middle and sort-last architectures.

In the sort-first strategy [6], the image space is partitioned into regions, called *tiles*, and each processor is responsible for all the rendering calculations (both geometry and rasterization) in the tiles to which it is assigned. Each 3D triangle is then distributed to the processor(s) that is responsible for the tile(s) with which that triangle overlaps. One triangle can be sent to multiple processors. Finally the image regions from the processors are simply combined together to form the rendered image.

Parallel rendering architectures have the disadvantage that the same triangle has to be sent to all the tiles with which it overlaps. As the tile size decreases, the overlap factor increases, and so does the communication requirement for triangle distribution. One solution to this problem is to use a compressed representation during the transfer that can significantly reduce the communication requirement. Note that as the viewpoint changes per frame, the set of triangles for each tile also changes. As a result, the compression algorithm should be run corresponding to all the tiles per frame. Unfortunately, current state-of-the-art triangle mesh compression algorithms are too computation intensive to be performed on-the-fly per frame.

The solution to this problem is to compress the entire triangle mesh statically once. During sorting, the single compressed mesh is subdivided into multiple compressed submeshes — one per tile. But it is challenging to do this subdivision in compression domain, that is, without decompressing the entire mesh and then compressing the triangles for each tile separately from scratch. In fact, to the best of our knowledge, there is no algorithm that can perform a compression-domain sorting of triangle mesh.

We have previously developed a simple and highly ef-

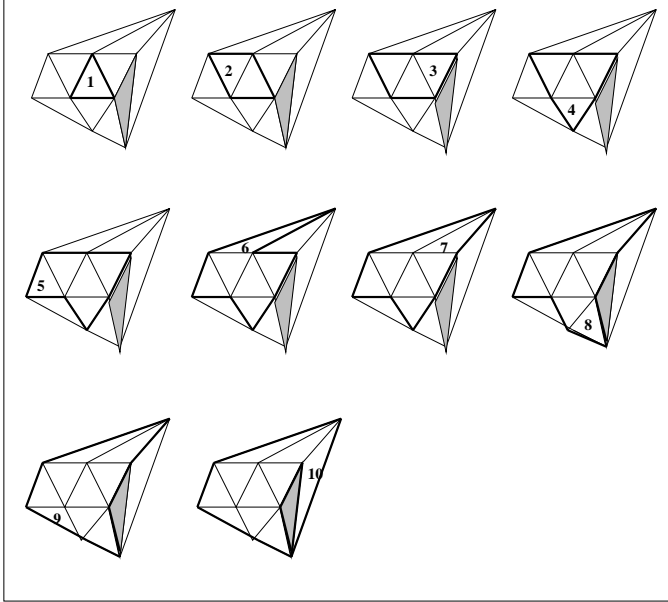


Figure 1. BFT mesh traversal.

efficient triangle mesh compression algorithm, called BFT mesh encoding [3, 4]. We have shown that that addition of a simple BFT decompressor at the front-end of the graphics processor can then directly accept a BFT mesh. In this work, we design a compression-domain sorting or distribution algorithm for BFT encoding. Our algorithm accepts a BFT mesh and the tile size as inputs. It then incrementally generates one BFT submesh per tile without explicit decompression and recompression. These BFT submeshes are then distributed to the parallel processors for decompression and rendering. This compression-based technique significantly reduces the communication bandwidth requirement during triangle distribution. To the best of our knowledge, this work is the *first attempt* towards on-the-fly sorting of compressed mesh representation.

The rest of the paper is organized as follows. Section 2 briefly describes BFT mesh compression algorithm. In Section 3, we discuss on-the-fly sorting of BFT mesh. Section 4 evaluates the performance of BFT-oriented parallel rendering system and Section 5 concludes the paper.

2 BFT Compression Algorithm

In this section, we briefly describe the *Breadth-First Traversal (BFT)* [4, 3] algorithm for triangle mesh compression. A triangle mesh is represented with *geometry* (a set of vertex positions, color, and other attributes) and *connectivity* (the incidence relations among vertices, edges, and triangles). Traditionally, each triangle in a triangle mesh

is represented independently in terms of the geometry of its three vertices. However, the overwhelming size of traditional triangle mesh representation has led to sophisticated, triangle-mesh-specific compression/decompression algorithms [2, 4, 7, 8, 9].

Triangle mesh compression consists of (1) lossless connectivity compression and (2) lossy geometry compression. BFT is a connectivity compression algorithm. It does not perform geometry compression. However, any efficient geometry compression algorithm can be easily integrated with BFT algorithm. BFT achieves a compression efficiency comparable to state-of-the-art mesh compression algorithms. At the same time, the simplicity of the BFT algorithm lends itself amenable to compression-domain sorting for parallel rendering architecture.

The basic idea of the BFT algorithm is to traverse a triangle mesh in a breadth-first order from a chosen *seed triangle*. The vertices of the seed triangle form a *frontier*. A frontier is a circular buffer of vertices. BFT visits each edge — consisting of two consecutive vertices — of the frontier and enumerates the unvisited triangle, if any, that is incident on that edge in terms of the *third vertex*. At the same time, it incrementally modifies the frontier to delete the vertices whose incident triangles have all been visited, and to add the new vertices. BFT continues to enumerate the triangles and modify the frontier till either there is only one vertex left in the frontier, or a frontier left with n vertices has not been modified for n consecutive steps. Figure 1 illustrates this traversal process with a small triangle mesh. The shaded portion in the figure is a hole in the triangle mesh. The bold lines indicate the frontier. The ordering of the triangles represents the order in which the triangles are enumerated.

The edge for which BFT attempts to find an incident and not-yet-visited triangle is called *current edge* and the two vertices of the edge, in order of their appearances in the frontier, are called *left vertex* and *right vertex*, respectively. A current edge for which BFT cannot find any unvisited triangle, because either it is a boundary edge or both of its incident triangles have been visited, is called a *null edge*.

The third vertex used to form a triangle with the current edge can be represented either explicitly in terms of its geometry or implicitly as a reference to some vertex that appeared previously. In case of BFT, this reference is a pointer into the frontier, specified as an offset from the right vertex or the left vertex, depending on which offset is smaller.

2.1 Encoding Commands

Given an input triangle mesh, BFT performs the following two steps: (1) it pre-processes the triangle mesh to find out the visiting order of the triangles; and (2) it represents the mesh as a command sequence, where each command encodes either a new triangle in terms of the corresponding

Before	After	Command
		NEW
		RF0
		LF0
		RF <4>
		LF <4>
		NULL
		DL
		DR

Figure 2. BFT encoding commands.

third vertex or the presence of a null edge. Figure 2 illustrates the different commands used by the BFT compression algorithm: first five commands encode the cases when a triangle is enumerated with a third vertex and the last three commands encode the different null edge cases. The left-hand and right-hand side of the figure represent the frontier before and after visiting the current edge $\{1,2\}$. The bold line indicates the current edge, and the broken lines are incident to the third vertex. We store geometry data for all the vertices separately as a vertex array sorted in the order in which they appear in the BFT mesh with *New* commands. The BFT decompression algorithm dynamically reconstructs the frontier of the BFT traversal and enumerates triangles on the frontier according to the information encoded in the command sequence.

3 Sorting of BFT Mesh

To take advantage of BFT mesh during triangle distribution, we have to generate on the fly a *tiled BFT mesh* — that is, one BFT submesh per tile — from the original BFT mesh. One technique to generate tiled BFT mesh works as follows. Suppose, we are given a clipping algorithm that can generate a BFT submesh corresponding to a particular tile by clipping the original BFT mesh against the tile bounding box. In that case, for n tiles, the system generates one BFT submesh corresponding to each tile. We will discuss how to speedup this process later in the paper. But first, let us describe the clipping algorithm that generates a

BFT submesh corresponding to a tile.

3.1 Clipping of BFT Mesh

Clipping removes the triangles outside a bounding box from the triangle mesh. This removal completely destroys the original triangle mesh structure. So we need to develop a new clipping algorithm for BFT mesh that can generate a clipped BFT mesh on the fly. The basic idea is as follows: as the original BFT mesh is decompressed, if a triangle is not clipped, we send the original triangle’s encoding as it is; if a triangle gets clipped, then we apply a set of *modification rules* to change or remove the encoding so as to remove the triangle in question from the mesh and send this new encoding.

The BFT mesh clipping algorithm uses a generic clipping algorithm (Cohen-Sutherland clipping algorithm [1]) to identify clipped and unclipped vertices and triangles. The BFT clipping algorithm then generates (1) a clipped BFT mesh, which contains only the unclipped triangles; and (2) a partially-clipped mesh, which is a set of independent triangles that are results of triangulation of the unclipped portion of the partially-clipped triangles.

The BFT clipping algorithm consists of a BFT mesh decompressor and a set of modification rules. When a triangle is decompressed, the algorithm first checks if the triangle is clipped or unclipped. The modification rules replace the corresponding BFT command such that the clipped vertices and the clipped triangles are deleted from the output BFT mesh. The new BFT command is stored in the output command stream. If a triangle is clipped, the algorithm uses per-triangle clipping to identify whether the triangle is completely or partially clipped. For partially-clipped triangles, the resulting triangulated unclipped portion is stored separately from the output BFT mesh. After traversing all the triangles in a BFT mesh, as the last step, the algorithm outputs all the independent triangles resulting from the triangulation of partially-clipped triangles.

3.1.1 Modification Rules

The goal of the modification rules is the following: If a triangle corresponding to a BFT command is clipped, then the BFT command should be modified such that the triangle is not present in the output clipped BFT mesh. Note that if all the triangle vertices are unclipped, then the triangle is also unclipped; the modification rules keep the BFT command unchanged in this case. But if any one of the triangle vertices is clipped, then the modification rules change the BFT command so as to remove the triangle. Recall that a BFT command not only encodes a triangle, but it also modifies the frontier. Let us assume that a BFT command is present in the input BFT mesh, but is modified in the clipped

v_l	v_r	v_x	t	c	c'
NC	NC	NC	NC	New/RF/LF	New/RF/LF
NC	NC	C	C	New/RF/LF	Null
NC	NC	NC	NC	RF0	RF0
NC	NC	C/-	C/-	RF0/DR	DR
NC	NC	NC	NC	LF0	LF0
NC	NC	C/-	C/-	LF0/DL	DL
NC	NC	-	-	Null	Null
C	C	NC	C	New/RF/LF	Add
C	C	C	C	New/RF/LF	ϕ
C	C	X/-	C/-	RF0/DR	ϕ
C	C	X/-	C/-	LF0/DL	ϕ
C	C	-	-	Null	ϕ
NC	C	NC	C	New/RF/LF	$\langle \text{Null}, \text{Add} \rangle$
NC	C	C	C	New/RF/LF	Null
NC	C	X/-	C/-	RF0/DR	ϕ
NC	C	X/-	C/-	LF0/DL	DL
NC	C	-	-	Null	Null
C	NC	NC	C	New/RF/LF	Add
C	NC	C	C	New/RF/LF	ϕ
C	NC	X/-	C	RF0/DR	DL
C	NC	X/-	C	LF0/DL	ϕ
C	NC	-	C	Null	ϕ

Table 1. Modification rules. v_l, v_r, v_x : left, right, and third vertex; t : triangle; c, c' : input and output command(s); **NC**: unclipped; **C**: clipped; **X**: don't care; **-**: undefined; ϕ : no output command.

BFT mesh. Then the frontier of the input BFT mesh, after processing the command, will differ from the frontier of the clipped BFT mesh. Because the vertices of the decompressed triangles depend on the state of the frontier, the rest of the commands will generate completely different triangles for the input and the clipped BFT mesh.

One solution to this problem is to maintain two frontiers during decompression and clipping of the BFT mesh: one corresponding to the input BFT mesh that is maintained by the decompressor, and another corresponding to the clipped BFT mesh generated so far. Now the modification rules can replace a command by looking forward and backward in both the frontiers starting from the current edge. However, the clipping algorithm is supposed to be fast to sustain interactive rendering rate, and hence cannot afford to perform such look-ahead.

3.1.2 Synchronization Invariants

Our idea is to design the modification rules such that the clipping algorithm can simulate the two logical frontiers with a single physical frontier. The modification rules

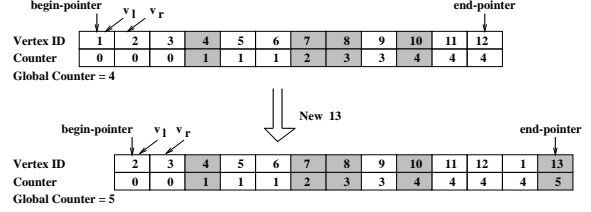


Figure 3. Global and local counters. The shaded vertices are clipped. The upper figure and lower figure show the frontier before and after the **New** command is processed, respectively. Vertex 13 is clipped and increases the global counter value by 1.

obey the following invariants to keep the logical frontiers “synchronized”, which allows us to replace a BFT command by only looking at the vertices that are involved in that command. The invariants are: (1) at any instant, the two frontiers are identical except that the clipped vertices are absent in the clipped BFT mesh’s frontier; and (2) at any instant, the current edges of the two frontiers are the same if the current edge is unclipped; otherwise the clipped BFT mesh’s current edge is the closest unclipped successor of the original mesh’s current edge. For example, if $\langle \{1, 2\}, 3, 4, 5, 6, 7, 8 \rangle$ is the original frontier with current edge $\{1, 2\}$ and the clipped vertices are marked in bold, then the corresponding frontier in the clipped BFT mesh is $\langle \{1, 3\}, 5, 6, 8 \rangle$ with $\{1, 3\}$ as the current edge. The BFT mesh clipping algorithm simulates the two frontiers by physically maintaining only the frontier of the original mesh and a status bit with each vertex in the frontier that indicates whether the vertex is clipped or not.

Table 1 shows the complete set of modification rules for different combinations of left, right, and third vertices. The formal proof of correctness that the set of modification rules maintains synchronization invariants and generates a semantically correct clipped BFT mesh can be found in [3].

3.1.3 Offset Modification

Note that we need to change the offset of RF and LF commands in the clipped BFT mesh. A naive method to generate the new offsets for RF and LF commands is to scan the frontier starting from the current edge till the third vertex is reached, and count the number of clipped vertices encountered during the scan. Let o be the original offset and c be the count of the number of clipped vertices. Then $o - c$ gives the new offset value.

This sequential scanning can be eliminated by maintaining a counter with each vertex in the frontier. The counter indicates the total number of clipped vertices encountered

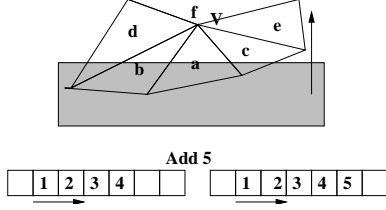


Figure 4. Add command. The shaded rectangle represents the clipped region. a,b,c,d,e,f is the triangle traversal order. Unclipped triangles e and f expect the vertex V to be present in the frontier. V is added to the frontier with Add command.

when that vertex was added at the end of the frontier. A global counter keeps track of the total number of clipped vertices encountered so far. Initially, the global counter is set to 0. When a vertex is added at the end of the frontier, the global counter is incremented by 1 if the vertex is clipped. The global counter value is attached to the counter field of the added vertex. Figure 3 illustrates the concept of the counter.

Now, for RF command with offset o , the count of the number of clipped vertices

$$c = \text{counter}[\text{begin-pointer} + 2 + o] - \text{counter}[\text{begin-pointer}],$$

and for LF command with offset o ,

$$c = \text{counter}[\text{end-pointer}] - \text{counter}[\text{end-pointer} - o]$$

For example, in Figure 3, if the triangle $\{1,2,5\}$ is represented as RF $\langle 2 \rangle$, then $c = 1$.

3.1.4 Add Command

The existing set of BFT commands is not sufficient to delete clipped triangles and maintain the two synchronization invariants. For example, consider a clipped triangle with clipped left and right vertices, but an unclipped third vertex. If the triangle is enumerated via New, LF, or RF command, then deleting the corresponding command from the clipped BFT mesh removes the triangle. But the first synchronization invariant requires the third vertex to be added to the frontier. The existing set of BFT commands does not allow to add a vertex without adding a triangle. Therefore, we need to extend the set of BFT commands with a new command, called Add to insert a vertex without encoding any triangle.

Add command is required for the unclipped vertices near the clipping plane that have both clipped and unclipped incident triangles. If the BFT traversal order visits a clipped

incident triangle before visiting the unclipped incident triangles, then the Add command is required to add the vertex to the frontier. This way the vertex can be used by the unclipped incident triangles. Figure 4 explains why Add command is required and how it changes the frontier.

Intuitively, the clipping algorithm keeps the BFT mesh intact till it encounters a clipped triangle. Then the algorithm repeatedly applies modification rules to take care of the triangles near the boundary. Next, BFT traversal reaches the portion of the triangle mesh outside the boundary, and the corresponding BFT commands are deleted. Finally, BFT traversal may again cross the boundary to come back to the unclipped portion. Figure 5 illustrates the BFT clipping algorithm with an example BFT mesh. The bold solid line in the figure indicates the frontier. The arrow represents the current edge while the circle represents the current triangle.

3.2 BFT Sorting

The sorting process uses 2D image-space coordinates of the triangle vertices to determine the tiles overlapping with the 2D image of the triangle. Once the submeshes for the tiles are constructed, they are sent to the rendering engines one BFT submesh at a time. We use the simple *bounding box sorting* algorithm to identify the tiles a triangle overlaps with. Bounding box algorithm constructs a screen aligned bounding box for each triangle (xmin, xmax, ymin, ymax) and finds the intersection of this bounding box with the tiles by simple comparisons. The triangle bounding box of a triangle t is denoted as $[t]$.

Given a BFT mesh and the tile size, we want to generate one BFT mesh per tile. A naive technique can generate a tiled BFT mesh by using the clipping algorithm for the BFT mesh as a subroutine. For n tiles, this technique requires n passes, where each pass sets the triangles corresponding to one tile as unclipped and all the others as clipped. Then in each iteration, the system generates a BFT mesh corresponding to one tile. The major disadvantage of this naive approach is that it requires the original mesh to be decompressed n times. To speedup the process, we want an algorithm that can modify the BFT submeshes corresponding to all the tiles as each triangle of the original mesh is decompressed.

One possible solution is to start with empty BFT meshes for all the tiles. We define a vertex v as unclipped for tile b , if v 's coordinate is inside tile b , and v is defined as clipped for all the other tiles. A triangle is unclipped or associated with a tile if all its vertices are unclipped for that tile. Now, as each BFT command is decompressed, the modification rules examine the clipping status of the triangle vertices with respect to each tile. The rules then generate modified (possibly empty) BFT command sequence for each of

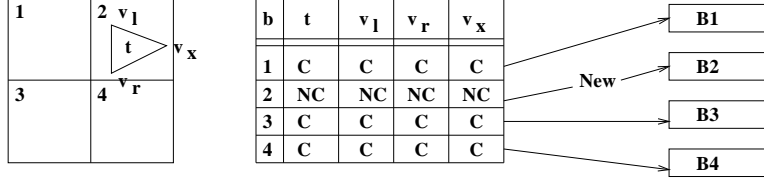


Figure 6. Sorting of BFT mesh. Image space is divided into four tiles in the left, 1–4, and the corresponding BFT meshes are B1–B4. Triangle t is enumerated in the original BFT mesh with `New` command and t is associated with tile 2. The command `New` is appended to B2, but no commands are appended to B1, B3, or B4.

the tiles and append these modified commands to the corresponding BFT meshes generated so far. Figure 6 illustrates how this solution works.

Unfortunately, this simple algorithm does not take care of the triangles that span multiple tiles. All the vertices of a shared triangle do not belong to one tile. Therefore, for any tile associated with a shared triangle, one of the triangle vertices will always be clipped for that tile. Hence, the modification rules consider this triangle as clipped with respect to all the associated tiles.

The fundamental problem is that the modification rules associate a vertex only with the tile for which it is unclipped; whereas sorting requires a vertex to be associated with all the tiles with which its incident triangles overlap. To solve the problem, we use the concept of vertex bounding box. Intuitively, a vertex bounding box $[v]$ of a vertex v includes all the tiles associated with v 's incident triangles. For a vertex v , $[v]$ is defined as a bounding box that is the smallest superset of all the incident triangles' bounding boxes. Figure 7 illustrates the concept of vertex bounding box.

A vertex v is unclipped for tile b , if $b \in [v]$ and clipped otherwise. Using a vertex bounding box ensures that the vertex is associated with all the tiles with which its incident triangles overlap. But because $[t] \subseteq [v_0] \cap [v_1] \cap [v_2]$, it is no longer guaranteed that for a tile b , if v_0, v_1 , and v_2 are all unclipped, then t is also unclipped for b . Therefore, the modification rules have to be changed such that a command enumerates a triangle t for tile b , if and only if $b \in [t]$.

Table 2 presents the modification rules to generate BFT meshes for all the tiles from the input BFT mesh. The second column shows the set of tiles for which the corresponding rule applies. For a particular input command c , the different rules cover mutually exclusive tiles. The next four columns show the status of the left vertex, right vertex, third vertex, and the triangle with respect to the tiles in the second column. Compare this table with Table 1 to make sure that they represent the same set of rules.

Sorting algorithm for the BFT mesh now consists of two passes. The first pass is a preprocessing step that scans

Dataset	Vertex	Triangle	Edge
Bunny	34,834	69,451	104,288
Horse	48,485	96,966	145,449
Hand	327,323	654,666	981,999
Dragon	437,645	871,414	1,309,256
Buddha	543,652	1,087,716	1,631,574
Blade	882,954	1,765,388	2,648,082

Table 3. Characteristics of mesh models.

through the command stream to calculate the vertex bounding box for each vertex in the vertex array. The second pass generates BFT meshes for all the tiles by applying modification rules to each command in the command stream. Experimental results show that the sorting of BFT mesh adds only about 15% overhead to the rendering pipeline as opposed to 10% for the sorting of independent triangles.

4 Performance Evaluation

In this section, we show the performance advantage of parallel rendering using BFT mesh as opposed to using independent triangles. We use six triangle-mesh based 3D models of varying complexity (refer Table 3).

Figure 8 shows the communication traffic for different tile sizes. Compared to *Tiled Tri*, *Tiled BFT* mesh cuts down the communication requirement by one-third. Notice that the percentage increase in communication traffic between sequential (*Tri*) and parallel rendering (*Tiled Tri*) with independent triangles is more than the percentage increase between sequential (*BFT*) and parallel rendering (*Tiled BFT*) with BFT. *Tri* representation uses a fixed storage per triangle and hence the difference between *Tri* and *Tiled Tri* is directly proportional to the ratio between the number of tiled triangles and original triangles. In case of BFT, the bandwidth is dominated by vertex geometry (16 bytes) and not the connectivity (typically 0.25 bytes per triangle). Therefore, the difference between *BFT* and *Tiled BFT* is proportional to the ratio between the number of tiled vertices and

c	Tile	v_l	v_r	v_x	t	c'
New/RF/LF	$[t]$	NC	NC	NC	NC	New/RF/LF
New/RF/LF	$([v_l] \cap [v_x]) - [t]$	NC	X	NC	C	$\langle \text{Null, Add} \rangle$
New/RF/LF	$([v_l] - [v_x]) - [t]$	NC	X	C	C	Null
New/RF/LF	$([v_x] - [v_l]) - [t]$	C	X	NC	C	Add
RF0	$[t]$	NC	NC	NC	NC	RF0
RF0	$([v_r] \cap [v_l]) - [t]$	NC	NC	X	C	DR
RF0	$([v_r] - [v_l]) - [t]$	C	NC	X	C	DL
DR	$[v_r] \cap [v_l]$	NC	NC	-	-	DR
DR	$[v_r] - [v_l]$	C	NC	-	-	DL
LF0	$[t]$	NC	NC	NC	NC	LF0
LF0	$[v_l] - [t]$	NC	X	X	C	DL
DL	$[v_l]$	NC	X	-	-	DL
Null	$[v_l]$	NC	X	-	-	Null
Other combinations						ϕ

Table 2. Modification rules to replace a BFT command with a possibly empty sequence of BFT commands to generate tiled BFT meshes. v_l : left vertex; v_r : right vertex; v_x : third vertex; t : triangle; c : input command; c' : output commands; NC: unclipped vertex; C: clipped vertex; X: don't care; -: undefined; ϕ : no command.

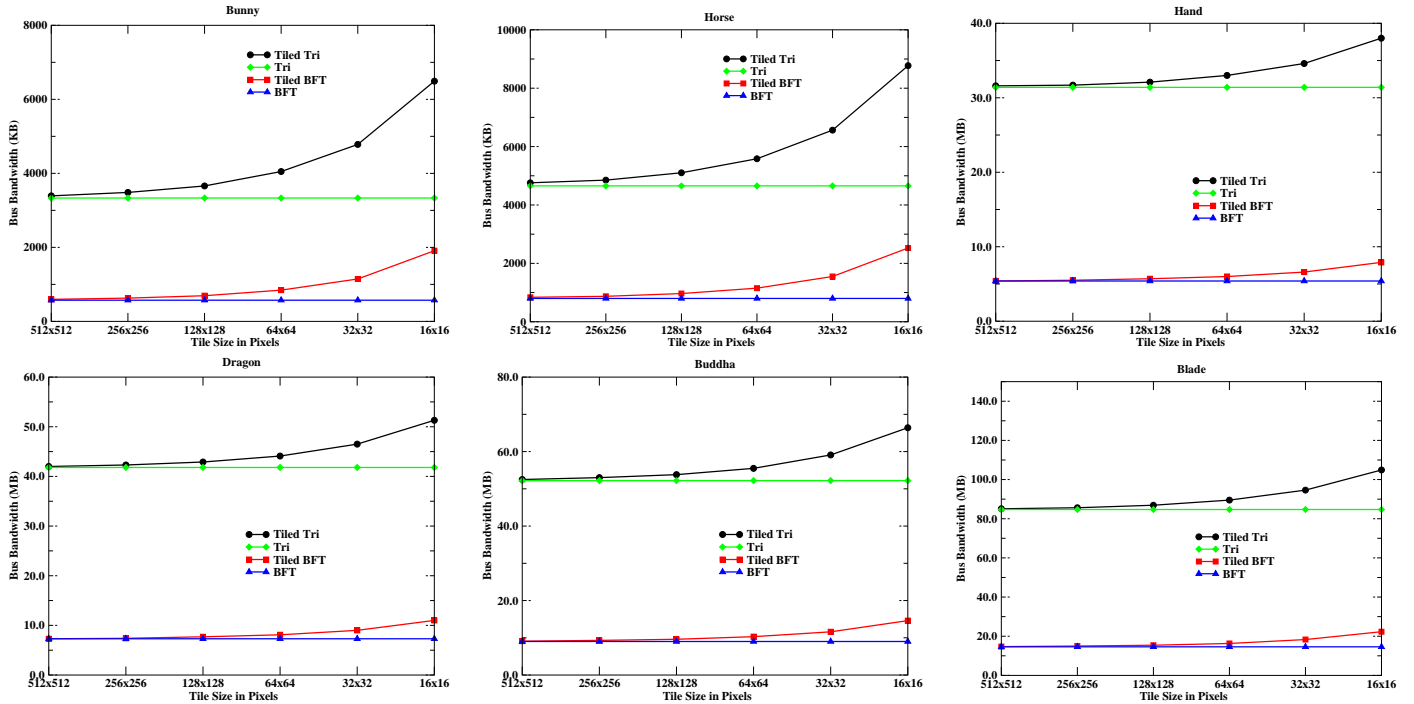


Figure 8. Communication requirement for parallel rendering with different tile sizes.

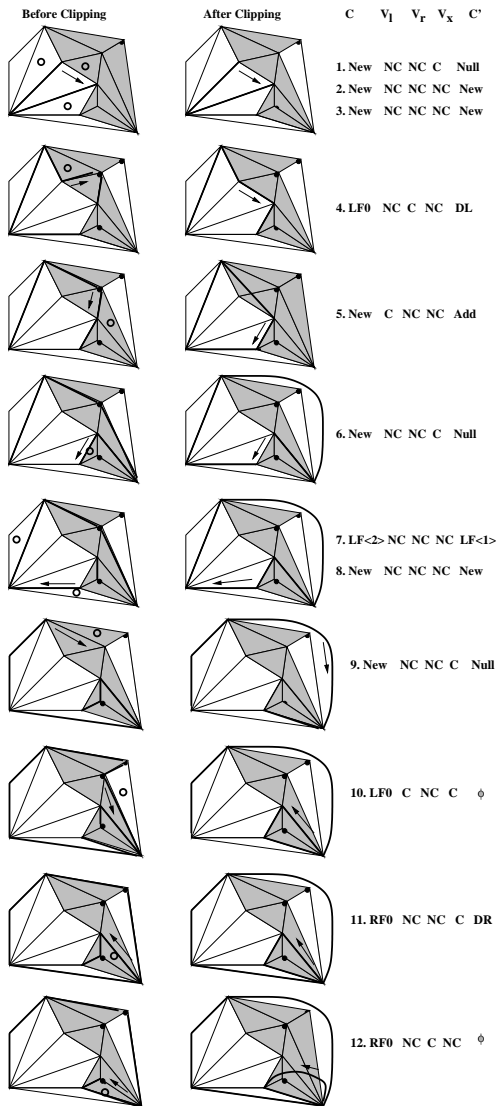


Figure 5. An example illustrating the BFT clipping algorithm. White triangles are unclipped and shaded triangles are clipped. Clipped vertices are marked with a square.

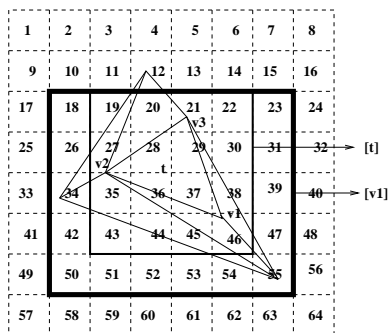


Figure 7. Bounding box of a vertex.

original vertices, which is smaller than the ratio for triangles. Hence, the percentage difference between *BFT* and *Tiled BFT* is smaller than the percentage difference between *Tri* and *Tiled Tri*.

5 Conclusion

Compression-domain processing of 3D mesh is a promising approach towards solving the memory capacity and bandwidth problems associated with large 3D meshes. This paper demonstrates the practicability of this approach for parallel rendering with prototype implementation. Performance evaluation of our approach suggests that compression-domain 3D mesh processing can significantly reduce the communication bandwidth requirement in the parallel rendering pipeline, thereby enabling a parallel graphics system to render very large 3D meshes that was not possible with traditional uncompressed approaches.

6 Acknowledgment

This research was supported by the following grants: NSF MIP-9710622, NSF ACI-9907485, and a grant from Sandia National Laboratory.

References

- [1] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice, Second Edition* in C. Addison Wesley, 1990.
- [2] S. Gumhold and W. Straßer. Real Time Compression of Triangle Mesh Connectivity. *Proceedings of SIGGRAPH 98*, pages 133–140, July 1998.
- [3] T. Mitra. *Mesh Compression and Its Hardware/Software Applications*. PhD thesis, Computer Science Department, SUNY Stony Brook, <http://www.ecsl.cs.sunysb.edu/tr/TR90.ps.Z>, December 2000.
- [4] T. Mitra and T. Chiueh. A Breadth-First Approach to Efficient Mesh Traversal. *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 31–38, August 1998.
- [5] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):231–240, July 1992.
- [6] C. Mueller. The sort-First Rendering Architecture for High-Performance Graphics. In *Proceedings of the ACM Symposium on Interactive 3D Graphics*, 1995.
- [7] J. Rossignac. Edgebreaker: Connectivity Compression for Triangle Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, January - March 1999.
- [8] G. Taubin and J. Rossignac. Geometric Compression Through Topological Surgery. *ACM Transactions on Graphics*, 17(2):84–115, April 1998.
- [9] C. Touma and C. Gotsman. Triangle Mesh Compression. *Graphics Interface '98*, pages 26–34, June 1998.