An FPGA Implementation of Triangle Mesh Decompression

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

This paper presents an FPGA-based design and implementation of a three dimensional (3D) triangle mesh decompressor. Triangle mesh is the dominant representation of 3D geometric models. The prototype decompressor is based on a simple and highly efficient triangle mesh compression algorithm, called BFT mesh encoding [10, 11]. To the best of our knowledge, this is the first hardware implementation of triangle mesh decompression. The decompressor can be added at the front-end of a 3D graphics card sitting on the PCI/AGP bus. It can reduce the bandwidth requirement on the bus between the host and the graphics card by up to 80% compared to standard triangle mesh representations. Other mesh decompression algorithms with comparable compression efficiency to BFT mesh encoding are too complex to be implemented in hardware.

1 Introduction

Three dimensional triangle mesh is the dominant representation of 3D geometric models. However, explosive growth in the complexity of mesh-based 3D models overwhelms the storage, transmission, and computing capability of existing graphics subsystems. In particular, one of the major bottlenecks of 3D graphics architectures is the bandwidth available on the system bus connecting the host CPU and the graphics processor [9, 12]. For a large triangle mesh, the bandwidth required by current triangle mesh representations may exceed the bandwidth available between the host CPU and the graphics processor. For example, the host processor requires about 38MB per frame to transfer the “Blade” model (in Table 1) using the standard triangle representation. In addition, the host processor may need to transfer texture images to the graphics processor. But, the two high-speed standards available for computer system bus, PCI and AGP, support peak bandwidth of only 528MB and 1024MB per second. In practice, the sustained bandwidth available is much less, which makes interactive rendering at thirty frames per second quite difficult, if not impossible.

An effective solution to the problems with large 3D meshes is to compress the mesh as much as possible on the host processor and to send the compressed mesh to the graphics processor. In fact, in recent years, a considerable amount of research efforts have been spent in developing efficient compression/decompression algorithms for 3D meshes. However, the focus of the previous research has been oriented mainly towards minimizing the size of the compressed mesh to save the network bandwidth or disk storage capacity. The issue of using compressed 3D mesh in the graphics processor in order to reduce PCI/AGP bus bandwidth requirement is largely unexplored. This is because rendering with compressed mesh requires decompression in hardware and the current triangle mesh compression algorithms are too complex to be decompressed in hardware. We have earlier developed an efficient mesh compression/decompression algorithm, called Breadth First Traversal (BFT) mesh encoding [10, 11]. In this work, we present the design and implementation of the first (to the best of our knowledge) FPGA-based hardware prototype for mesh decompressor. This decompressor can be added at the front-end of a graphics processor so as to render a compressed mesh encoding and thereby reduce the bandwidth requirement between the host and the graphics processor by as much as 80% compared to existing mesh representations.

2 Related Work

A triangle mesh is a piecewise linear surface, consisting of a set of triangular faces, such that any two faces either are disjunctive or share an edge or a vertex. A triangle mesh is represented with *geometry* (a set of vertex positions, color, normal, texture, 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. Assuming 32 bytes of geometry information per vertex, the *independent triangles* representation requires $32 \times 3 = 96$ bytes per triangle. An improvement over this representation is *indexed independent triangles* which consists of a *vertex array* containing the geometry information for all the vertices, plus a set of triangles, each represented by three indices (each 4 bytes) into the vertex array. However, for a large triangle mesh, the entire vertex array cannot be stored in either on-chip or off-chip memory of a graphics processor; it is stored in host processor memory. When the graphics processor receives a vertex index, it has to fetch the corresponding geometry information from host memory over the system bus. This makes the total bus bandwidth requirement as high as 108 bytes per triangle.

To reduce this bandwidth requirement, virtually all commercial graphics processors support a more succinct representation, called *triangle strip*. It encodes a sequence of triangles such that every triangle, except for the first, shares an edge with its immediate predecessor. Therefore, except for the first triangle, all others can be represented by one vertex each. The graphics processor deploys a two-entry vertex buffer for temporary storage of the shared edge. Deering [2] extended this idea to a 16-entry FIFO vertex buffer, so that an already visited vertex need not be respecified if it exists in the buffer. This is known as *generalized triangle mesh* and it can potentially encode a mesh by specifying each vertex exactly once. Chow [1] proposed various algorithms to construct generalized triangle meshes so as to maximize the vertex buffer hit rate. Finally, Hoppe [7] proposed *transparent vertex cache* that is similar to a general-purpose processor cache. Vertex cache stores vertex index as tag, geometry as data, and achieves hit rate similar to the vertex buffer for generalized triangle mesh. However, none of these techniques can achieve bandwidth reduction comparable to BFT as we will see in Section 5.

The overwhelming size of traditional triangle mesh representation has also lead to sophisticated, triangle mesh specific compression/decompression algorithms. Triangle mesh compression consists of (1) lossless connectivity compression and (2) lossy geometry compression. In this paper, we will only concentrate on connectivity compression. Geometry compression uses quantization and predictive encoding, which are well studied solutions in image processing with highly efficient hardware-based implementation of decompression logic. Connectivity compression algorithms on the other hand are purely software based techniques [15, 6, 13, 16] aimed to reduce the network bandwidth and storage requirements for triangle meshes. They opt for higher compression efficiency, which leads to decompression algorithms that are too complex to be easily implemented in hardware. As a result, we need to use highly compressed 3D models during download and convert them to not so efficient representations such as triangle strip or generalized triangle mesh for rendering. In contrast, BFT combines high mesh compression efficiency with hardware-amenable decompression scheme that makes it a true end-to-end triangle mesh representation.

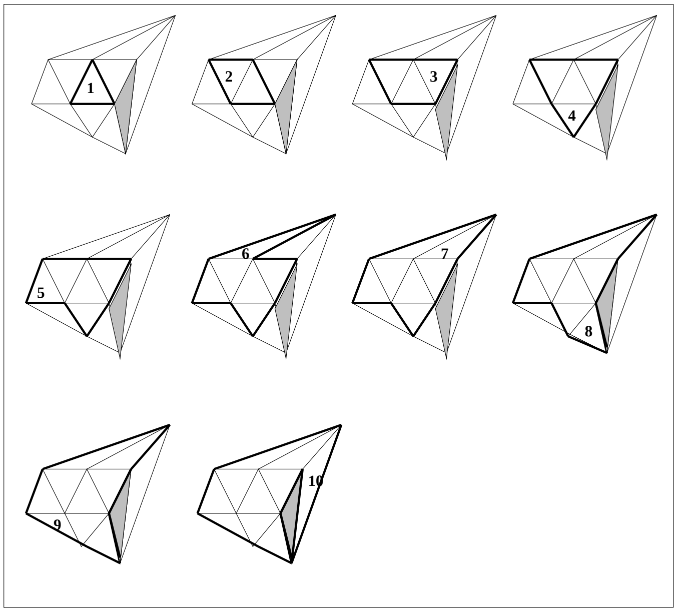


Figure 1. BFT mesh traversal.

3 BFT Mesh Compression

In this section, we briefly describe the *Breadth-First Traversal (BFT)* [11, 10] algorithm for triangle mesh connectivity compression. BFT does not perform geometry compression. However, any efficient geometry compression algorithm can be easily integrated with BFT algorithm. The mesh decompressor, in that case, should include a decoder for predictive-encoded geometry data as shown in [2].

The basic idea of 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

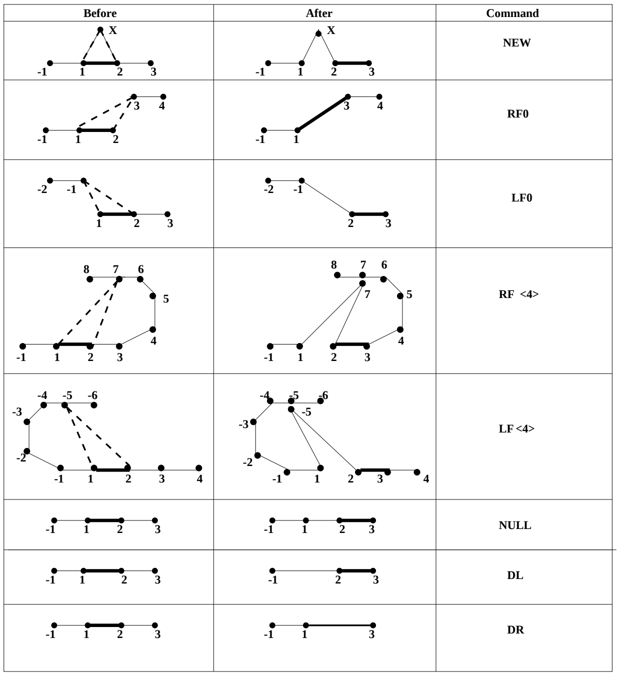


Figure 2. BFT encoding commands.

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.

3.1 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 encodes the mesh as a command sequence, where each command encodes either a new triangle in terms of the corresponding third vertex or the presence of a null edge. Figure 2 illustrates the different commands used by BFT compression algorithm: first five encode the cases when a triangle is enumerated with a third vertex and the last three 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.

1. `New-Vertex (New):` BFT enumerates a new triangle by pairing up the current edge with a third vertex, which is represented explicitly. This command adds the third vertex to the frontier, thereby deleting the current edge and adding two new edges to the frontier. The current edge moves to the next edge in the frontier.

2. `Right-Frontier-0 (RF0):` BFT enumerates a new triangle by pairing up the current edge with the immediate neighbor of the right vertex. This is a case of 0 offset value for the pointer. The right vertex is deleted, which adds a new edge between the left and the third vertex.

3. `Left-Frontier-0 (LF0):` This command is a mirror of the previous command.

4. `Right-Frontier` $\langle$`offset`$\rangle$ `(RF):` BFT enumerates a new triangle by pairing up the current edge with a third vertex that is implicitly represented with an offset value greater than 0 for the pointer. The offset value gives the distance of the third vertex from the right vertex. This command modifies the frontier in a similar fashion as `New` command.

5. `Left-Frontier` $\langle$`offset`$\rangle$ `(LF):` This command is a mirror of the previous command.

6. `Null (Null):` There is no unvisited triangle incident on the current edge. However, both the right- and left vertex have some unvisited incident triangles. The frontier is kept as it is and the current edge moves to the next edge in the frontier.

7. `Delete-Left-Vertex (DL):` This command deletes the left vertex. It is used if the left vertex has no unvisited incident triangles. This command modifies the frontier exactly in the same manner as `LF0` command, but does not enumerate any triangle.

8. `Delete-Right-Vertex (DR):` This command is a mirror of the previous command.

The compressed triangle mesh should contain geometry information in addition to BFT-encoded connectivity information. 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.

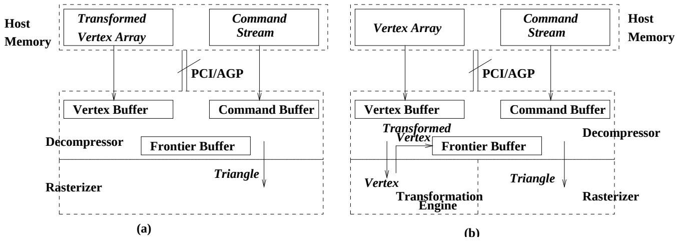


Figure 3. BFT rendering pipeline.

4 BFT Mesh Decompressor

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. This section describes the mapping of the BFT mesh decompression algorithm to an FPGA implementation. But first we describe how the decompression process is integrated with 3D graphics rendering pipeline.

4.1 Integrating Decompression in Rendering

The 3D graphics rendering pipeline consists of two distinct stages: *geometric transformation* and *rasterization* [5]. The geometric transformation stage maps triangles from a 3D coordinate system (object space) to a 2D coordinate system (image space). The operations in geometric transformation stage are per-vertex operations (i.e., they require only the vertex array and not the command stream); where the operations in rasterization stage are per-triangle operations (i.e., they require both the vertex array and the command stream). Currently, the rasterization stage is almost always implemented in dedicated graphics processor sitting on the PCI/AGP bus. The geometric transformation stage on the other hand can either be implemented in the graphics processor or as a software running on the host processor. In both cases, a major concern is the bandwidth required to transfer the triangle mesh (transformed or untransformed) from the host to the graphics processor. This bandwidth requirement is reduced by using a BFT-mesh representation, which in turn requires a hardware BFT mesh decompressor as the front end of the graphics processor.

Figure 3 illustrates how the BFT mesh decompressor is integrated into the rendering pipeline. BFT mesh consists of a vertex array and a command stream. If geometric transformation is implemented in software, it operates on the vertex array to create a transformed vertex array, and stores the command stream and the transformed vertex array in host memory. The graphics processor consists of a rasterizer and a front-end decompressor. The decompressor DMAs the transformed vertex array and the command stream over the PCI/AGP bus into its vertex- and command buffer, converts the command stream with the help of the frontier buffer into independent triangles, and sends the resulting triangles to the rasterizer. The decompression process is completely transparent to the rasterizer.

If geometric transformation is implemented in hardware, the host processor simply puts the vertex array and the command stream in host memory so that the mesh decompressor can DMA them into its own buffers. The graphics processor in this case consists of a transformation engine, a rasterizer, and the decompressor at the front-end. Vertices are not transformed when they reach the graphics processor. The decompressor sends a vertex buffer to the transformation engine, receives back the corresponding transformed vertex buffer, and then generates decompressed triangles for the rasterizer. Again the decompression process is completely transparent to both geometric transformation and rasterization.

4.2 Architectural Design

BFT mesh encoding is designed to lend its decompression algorithm to direct hardware implementation because (1) BFT mesh decoding logic is simple, and (2) BFT decoding accesses the frontier of BFT traversal in a sequential, localized, and completely predictable fashion. Even though BFT requires a relatively large frontier buffer (16KB–64KB for the test 3D models), the sequential access pattern permits perfect prefetching of the required portion of the frontier. Therefore, BFT decompressor rarely needs to wait for a frontier buffer access to complete.

The FPGA-based decompressor consists of a frontier buffer, which is equal to the maximum size of the frontier during compression/decompression, and two pointers, the begin-pointer and end-pointer, the entries between which represent the frontier of BFT traversal during the decompression process. Logically the frontier buffer is circular — that is the begin-pointer is the successor of the end-pointer, and the end-pointer is the predecessor of the begin-pointer. Current edge is represented by the begin-pointer and its successor. As the command for current edge is processed, the frontier is modified. Figure 4 illustrates how the different BFT commands modify the frontier and maintain the semantic of the commands as explained in Figure 2.

Implementing the frontier as a linked list in hardware is expensive because, when a vertex is deleted or added in the middle of the frontier, the frontier buffer entries need to be shifted in order to create or fill up the space for that vertex. To avoid this problem, we append the vertices with unvisited incident triangles after the end-pointer. The vertices that do not have any unvisited incident triangle are deleted from the frontier by simply advancing the begin-pointer to the next entry. With this mechanism, the begin-pointer and

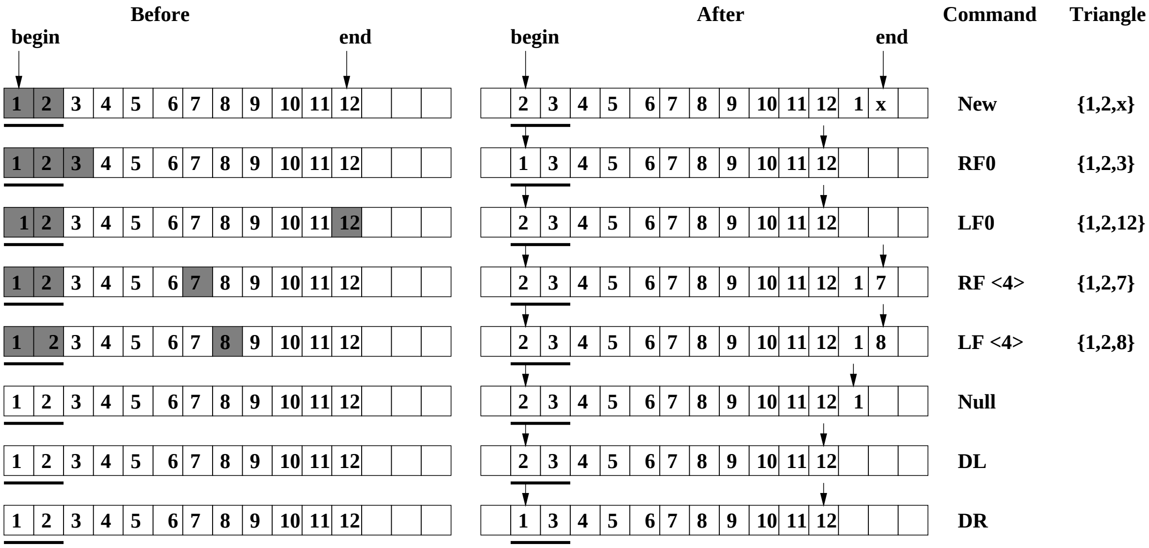

Before
After
Command
Triangle
begin
end
begin
end
1 2 3 4 5 6 7 8 9 10 11 12 | 2 3 4 5 6 7 8 9 10 11 12 1 x | New | {1,2,x}
1 2 3 4 5 6 7 8 9 10 11 12 | 1 3 4 5 6 7 8 9 10 11 12 | RF0 | {1,2,3}
1 2 3 4 5 6 7 8 9 10 11 12 | 2 3 4 5 6 7 8 9 10 11 12 | LF0 | {1,2,12}
1 2 3 4 5 6 7 8 9 10 11 12 | 2 3 4 5 6 7 8 9 10 11 12 1 7 | RF <4> | {1,2,7}
1 2 3 4 5 6 7 8 9 10 11 12 | 2 3 4 5 6 7 8 9 10 11 12 1 8 | LF <4> | {1,2,8}
1 2 3 4 5 6 7 8 9 10 11 12 | 2 3 4 5 6 7 8 9 10 11 12 1 | Null
1 2 3 4 5 6 7 8 9 10 11 12 | 2 3 4 5 6 7 8 9 10 11 12 | DL
1 2 3 4 5 6 7 8 9 10 11 12 | 1 3 4 5 6 7 8 9 10 11 12 | DR

Figure 4. FPGA-based decompression of BFT commands. The frontier is a circular buffer represented by the vertices between begin and end arrow. The bold solid line is drawn below the current edge. The shaded vertices participate in forming a triangle.

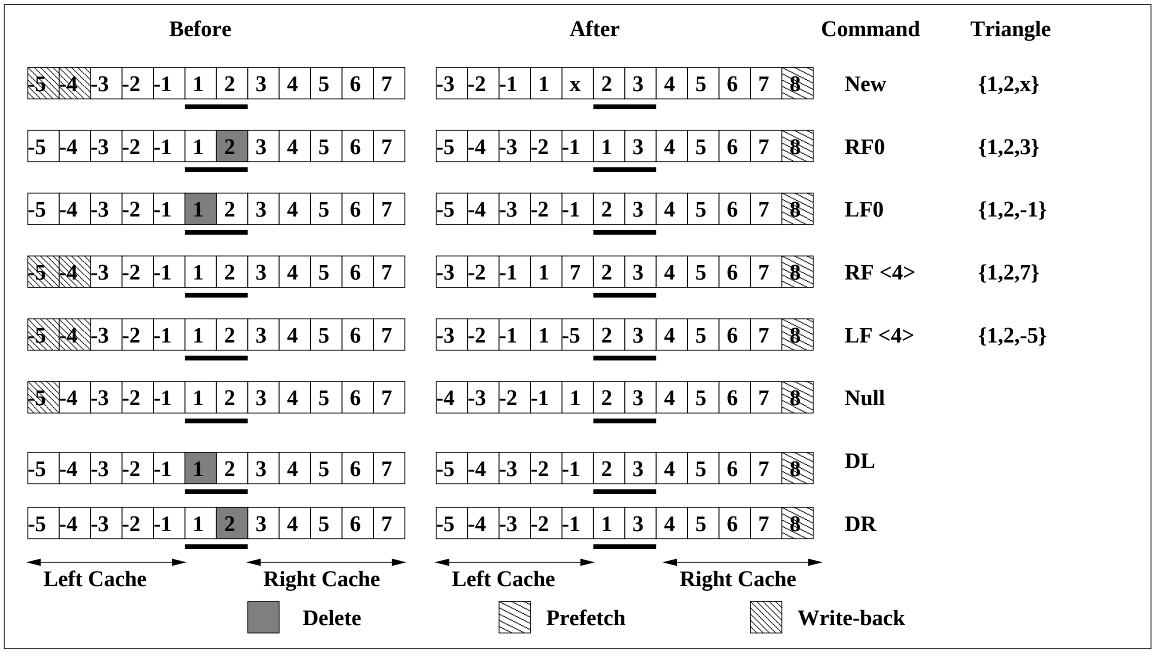

Before
After
Command
Triangle
-5 -4 -3 -2 -1 1 2 3 4 5 6 7 | -3 -2 -1 1 x 2 3 4 5 6 7 8 | New | {1,2,x}
-5 -4 -3 -2 -1 1 2 3 4 5 6 7 | -5 -4 -3 -2 -1 1 3 4 5 6 7 8 | RF0 | {1,2,3}
-5 -4 -3 -2 -1 1 2 3 4 5 6 7 | -5 -4 -3 -2 -1 2 3 4 5 6 7 8 | LF0 | {1,2,-1}
-5 -4 -3 -2 -1 1 2 3 4 5 6 7 | -3 -2 -1 1 7 2 3 4 5 6 7 8 | RF <4> | {1,2,7}
-5 -4 -3 -2 -1 1 2 3 4 5 6 7 | -3 -2 -1 1 -5 2 3 4 5 6 7 8 | LF <4> | {1,2,-5}
-5 -4 -3 -2 -1 1 2 3 4 5 6 7 | -4 -3 -2 -1 1 2 3 4 5 6 7 8 | Null
-5 -4 -3 -2 -1 1 2 3 4 5 6 7 | -5 -4 -3 -2 -1 2 3 4 5 6 7 8 | DL
-5 -4 -3 -2 -1 1 2 3 4 5 6 7 | -5 -4 -3 -2 -1 1 3 4 5 6 7 8 | DR
Left Cache
Right Cache
Left Cache
Right Cache
Delete
Prefetch
Write-back

Figure 5. FPGA-based decompression of BFT commands with prefetching. The bold solid line is drawn below the current edge. Left- and right cache consist of 5 entries each. The left- and right side of the figure show the caches before and after processing the command.

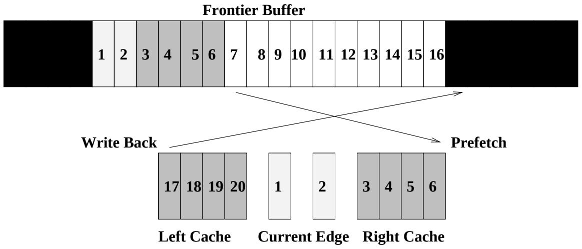


Figure 6. Prefetching of frontier buffer.

its successor always represent the current edge, addition and deletion of vertices always take place at the begin-pointer or end-pointer, and no shifting of the frontier buffer entries is necessary.

For all the BFT commands, the begin-pointer advances by one, and the end-pointer advances by zero to two entries at a time. When the end-pointer reaches the end of the frontier buffer, it is wrapped around to the beginning. The frontier buffer size is chosen as the smallest 2's power that is greater than or equal to the maximum frontier size. This greatly simplifies the modulo operation for wrap around from integer division to integer shift.

4.2.1 Prefetching of Frontier Buffer

The maximum size of the frontier buffer for BFT encoding is proportional to the width of the breadth-first traversal tree of the input mesh. For large triangle meshes, the size of the frontier buffer may become too large to be maintained in on-chip cache, thus leading to longer execution time than expected due to off-chip memory access overhead. Fortunately, BFT mesh decoding visits the vertices in a perfectly predictable way, and therefore perfect prefetching of the frontier buffer is possible. Also, the frontier buffer is small enough to be stored on board (i.e., on the FPGA board) so that BFT decompressor does not incur the extremely high latency of fetching data from host memory over PCI/AGP bus.

The key observation behind frontier buffer prefetching is that the third vertex, if represented as a reference to the frontier buffer, falls within a small distance from the current edge. As a result, only a small window of vertices around the current edge needs to be present on chip. The small window is organized as two FIFO caches, one for the left neighbors of the current edge and one for the right neighbors, as illustrated in Figure 6. The left cache holds the vertices visited in the recent past and the right cache holds the vertices to be visited in the near future. This window size is a small percentage of the total frontier buffer size. Moreover, the fact that the current edge moves in a predictable fashion through the frontier implies that this window can be prefetched perfectly. Maintaining only the small active window on chip incurs a performance overhead. When the third vertex falls out of the active window, it has to be brought in from the off-chip frontier buffer, which will incur the latency of off-chip memory access. Fortunately, most vertices fall within the small active window. Figure 5 illustrates how various BFT commands modify the two caches. All the BFT commands require prefetching of one entry from off-chip memory. `New`, `RF`, and `LF` require write-back of two entries, whereas `Null` requires write-back of one entry to off-chip memory. `RF0`, `LF0`, `DR`, and DL on the other hand do not require any write-back. For most 3D models, `New` and `RF0/LF0` constitute about 50% and 45% of all the commands, respectively. Therefore, the BFT decompressor requires a sustained rate of one memory read and one memory write operation per triangle. If the time to rasterize a triangle is longer than the time to perform one read and one write from off-chip memory, then the read/write delay can be completely masked. A rasterizer with peak performance of 20 million triangles/sec requires about 50ns to rasterize one triangle. Assuming 16 bytes of quantized geometry information per vertex (6 bytes for three coordinates, 6 bytes for three normals, and 4 bytes for color), the BFT decompressor requires 32 bytes per 50ns or 640MB/sec of peak memory bandwidth for frontier memory access. Because the decompressor accesses the frontier in sequential fashion, current generation memory that are optimized for sequential access instead of random access (e.g., 133 MHz SDRAM with peak bandwidth of 1064MB/sec) can easily support the bandwidth requirement of the decompressor.

4.2.2 Very Large Frontier Buffer

An FPGA decompressor can have only a fixed amount of off-chip memory for the frontier buffer. The experimental evaluation section (see Table 5) shows that BFT requires 16KB–64KB of frontier buffer for 3D models whose size ranges from 40,000 to 1.8 million triangles. A 64-KB frontier buffer can be easily fit into on-chip cache, not to mention on-board memory. But there is no theoretical bound on the maximum frontier size for very large triangle meshes, because maximum frontier size is proportional to $\sqrt{v}$, where v is the number of vertices. For our test triangle meshes, maximum frontier size varies from $2.4\sqrt{v}$–$4.3\sqrt{v}$. Therefore, it is possible that a 3D model's maximum frontier size exceeds the amount of available off-chip memory. One solution to this problem is to use the host memory to store the entire frontier buffer and the on-board memory on the graphics accelerator as a cache. But this solution will require as much bandwidth between the host and the graphics processor as traditional representations, thus defeating the purpose of BFT mesh.

To address this problem, we modify the BFT compression program to accept the amount of available on-board

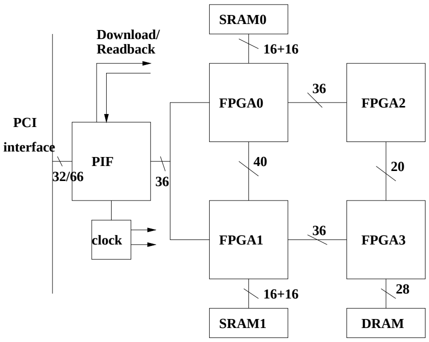


Figure 7. PCI Pamette architecture.

frontier memory as a parameter. If at any point during compression, the maximum frontier size exceeds the limit, the bounding volume of the model is divided into four parts. Then each part is compressed separately to generate four BFT meshes. If the maximum frontier size of any part again exceeds the limit, then that part is subdivided into four parts. This recursive subdivision continues till each part can be encoded as a BFT mesh without exceeding the frontier memory size limit. The vertices shared by two adjacent parts need to be duplicated. As a result, the compression efficiency of a set of BFT submeshes is less than a single BFT mesh. However, because shared vertices are only a small percentage of the input mesh vertices, the increase in size due to such mesh division is expected to be small. For our test triangle meshes, if the bounding volume is divided into four parts, then only 1% of the vertices are shared between the subparts. One can also use a more sophisticated mesh partitioning algorithm such as METIS [8].

4.3 Mapping to FPGA Implementation

4.3.1 Hardware and Software Environment

We implemented the hardware BFT decompressor using DEC's prototyping board called PCI Pamette [14]. PCI Pamette is a PCI card based on Xilinx 4000 series FPGA (XC4010E). PCI Pamette board has four user programmable FPGAs (FPGA0, FPGA1, FPGA2, and FPGA3) arranged in a 2×2 matrix. There is another FPGA that interfaces with the PCI bus and is known as PCI interface FPGA (PIF). The interface FPGA is loaded at power-on from a serial ROM. The host may then load the user FPGAs via the interface FPGA. In addition, the board has two banks of 64K × 16-bit SRAMs connected to FPGA0 and FPGA1, respectively, and allows 4MB–256MB of DRAM to be attached to FPGA3. Figure 7 illustrates how the different components are connected together. PCI Pamette uses the same clock frequency as the PCI clock (i.e., 33/66 MHz). But a programmable clock generator can use the PCI clock as reference and can produce any frequency from 400KHz to 100MHz.

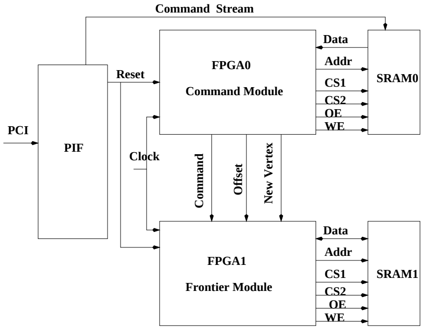
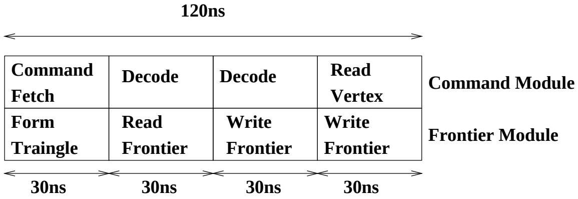


Figure 8. Block diagram of the decompressor.



Command Fetch	Decode	Decode	Read Vertex	Command Module
Form Traingle	Read Frontier	Write Frontier	Write Frontier	Frontier Module
30ns	30ns	30ns	30ns	

Figure 9. Decompressor pipeline stages.

We use Verilog hardware description language to describe the BFT decompressor and translate the high-level design, using Xilinx tools, to a format that can be directly downloaded into the FPGAs. In addition, PCI Pamette provides utilities that can directly transfer data between the host and SRAM0/SRAM1, and a controller that allows one to set the programmable clock generator and read/write output/input of the user programmable FPGAs via the PCI interface FPGA.

4.3.2 Datapath

The BFT decompressor prototype accepts the input BFT mesh from the PCI bus and generates independent triangles for the rasterizer module. The prototype decompressor can be tightly integrated with a rasterizer to create a 3D rendering pipeline that can directly accept BFT mesh.

The physical layout of the prototype BFT decompressor is shown in Figure 8. For ease of implementation, we download the command stream over the PCI bus and store it in SRAM0. The decompressor uses SRAM1 as the off-chip frontier memory. The decompression logic is distributed over the command module (FPGA0), which reads and decodes the variable-length BFT commands, and the frontier module (FPGA1), which maintains the left- and right cache, forms the triangles, and modifies the frontier and left/right caches for each processed BFT command. Because PCI Pamette provides a limited 16-bit width data bus

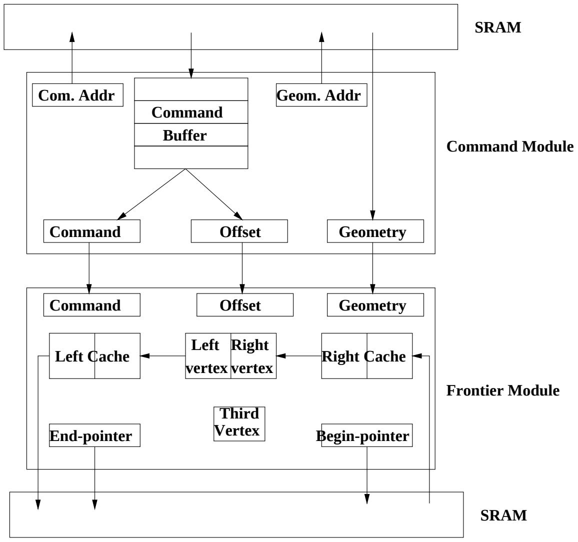


Figure 10. Detailed datapaths for command and frontier modules.

between SRAM and FPGA, the geometry information associated with a vertex is assumed to be 2-byte long for the prototype implementation. Current generation memories can provide 128-bit or 16-byte data bus to read/write the vertex data in one memory access. The left- and right cache are 2 entries each due to the limited amount of logic on each FPGA.

The decompressor is implemented as a simple pipeline consisting of two stages: (1) command fetch and decode stage and (2) frontier read/write stage. The first pipeline stage is implemented in the *command* module and the second pipeline stage is implemented in the *frontier* module. When the command module fetches and decodes command i, the frontier module modifies the frontier corresponding to command $i-1$. Figure 9 shows the substages in each of these two pipeline stages. BFT uses variable-length commands; therefore the command module maintains a small 4-byte command buffer. The command fetch sub-stage is executed to fetch 2 bytes from the command stream in SRAM0 only if there is space in the command buffer. The decode substage maps the received bit string to a BFT command and offset for RF and LF commands. If the BFT command is New then the read vertex sub-stage fetches the next entry from the vertex array.

The command, the offset, and the vertex (for New command) are sent to the frontier module in the next cycle. The frontier module forms the triangle and then modifies the frontier. The read frontier sub-stage is for prefetching from SRAM1 and the two write frontier substages are for write-back to SRAM1. If the third vertex is not present in the on-chip frontier cache, then the pipeline is stalled for one cycle to fetch the vertex from the off-chip frontier memory, i.e., SRAM1.

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 1. Characteristics of models.

Primitive	Total	Used	Percentage
CLB	800	723	90%
Flip-flop	1600	381	24%
4-input LUT	1600	1313	82%
3-input LUT	800	165	21%

Table 2. Area requirement of decompressor.

Figure 10 shows register-level details of the command and frontier modules. The begin-pointer and end-pointer are the pointers to the begin and end of the frontier in SRAM1. Command address and geometry address are the pointers to the command stream and the vertex array in SRAM0, respectively.

5 Performance Evaluation

In this section, we present the performance evaluation of the BFT mesh decompressor and demonstrate the performance advantage of integrating mesh decompression in the 3D rendering pipeline. We use six triangle-mesh based 3D models of varying complexity for experimental evaluation. Table 1 shows the characteristics of these datasets. The quantized geometry information associated with each vertex is 16 bytes (6 for coordinates, 6 for normals, and 4 for color). Using predictive encoding of the vertex geometry can further reduce the size of each vertex. Table 2 shows the number of different basic building blocks used in the FPGA implementation of the BFT mesh decompressor.

5.1 Decompression Performance

PCI Pamette runs at a maximum clock speed of 100MHz. At this clock rate, it requires 30ns to synchronously read or write from an on-board SRAM chip (Samsung KM68257C). Therefore the pipeline cycle (refer to 9) in the prototype implementation is chosen to be 120ns and the maximum achievable decompression rate with this prototype is 8.33 million triangles/sec. The actual decompression rate is smaller due to the presence of commands that

Dataset	Million Triangles/sec
Skull	7.867
Bunny	8.176
Horse	8.076
Hand	8.065
Dragon	7.785
Buddha	7.676
Blade	7.921

Table 3. Prototype FPGA decompression rate.

Dataset	Tri	Strip (% Tri)	BFT (% Tri)
Skull	1.920	0.931 (48.5)	0.331 (17.2)
Bunny	3.333	1.485 (44.5)	0.573 (17.2)
Horse	4.654	1.990 (42.8)	0.799 (17.2)
Hand	31.423	14.785 (47.0)	5.400 (17.2)
Dragon	41.827	20.336 (48.6)	7.259 (17.4)
Buddha	52.210	25.392 (48.6)	9.041 (17.3)
Blade	84.738	38.226 (45.1)	14.596 (17.2)

Table 4. Bandwidth required between host and graphics processor in MB.

do not form any triangle and the stall for cache misses. Table 3 shows the decompression rate for the test 3D models running on the FPGA prototype.

5.2 Bus Bandwidth Reduction

Table 4 shows the bandwidth required between the host and the graphics processor for three different triangle mesh representations: independent triangles, triangle strips, and BFT. Triangle strips are generated from the triangle mesh models using Stripe version 2.0 [3], which is a non-commercial software based on the stripification algorithm proposed in [4]. Triangle strip requires some extra bytes to distinguish one triangle strip from the next. We ignore that cost in our evaluation. On an average, the bandwidth requirement for triangle strip is 45% of that of independent triangles, and the bandwidth requirement for BFT is 17% of that of independent triangles. Again predictive encoding of the geometry data can further reduce BFT's bandwidth requirement by as much as five times.

Since the triangle strip generation code for transparent vertex caching [7] is not available, we cannot directly compare BFT with the vertex caching approach. However, vertex caching uses indexed triangle strip representation that requires at least one $log(v)$ bit vertex index per triangle (v is the number of vertices) for connectivity alone, compared to 1.86–2.53 bits per triangle for BFT connectivity. In addition, vertex caching has a miss rate of 0.65 per triangle, which translates to 30% higher bandwidth requirement compared to BFT.

Dataset	Frontier Vertices	Frontier Buffer	Vertex Array	%
Skull	605	16	320	5.00%
Bunny	541	16	557	2.87%
Horse	538	16	776	2.06%
Hand	1,650	32	5,237	0.61%
Dragon	2,148	64	7,002	0.91%
Buddha	2,383	64	8,698	0.73%
Blade	3,515	64	14,127	0.45%

Table 5. Storage requirement.

5.3 Frontier Buffer size

A major concern with the FPGA implementation of the BFT mesh decompressor is the size of the frontier buffer. The frontier buffer, if larger than the on-chip cache, can reside in off-chip memory; but it should still reside on-board — that is, local to the graphics card. Table 5 shows the maximum frontier size for different datasets in terms of the number of vertices and the amount of memory required for frontier buffer. Actual allocated frontier buffer size is equal to the smallest 2's power that is greater than or equal to the maximum frontier size. For all the test 3D models, the frontier buffer size is no more than 64KB, even when the input 3D model has close to million vertices. This implies that the frontier buffer can be easily stored in off-chip memory, if not on-chip. Compared to the vertex array scheme, Table 5 shows that BFT requires lower than 1% as much storage for large models.

5.4 Frontier Cache Hit Rate

The key claim of the proposed BFT scheme is that at any point of time, only a small window of the frontier buffer is needed to allow vertex reuse. As a result, the on-chip storage requirement is small and is independent of the input mesh size. Figure 11 demonstrates that this is indeed the case by showing the third vertex hit rate versus the cache size. If the cache size is N, then the third vertex reference is a hit if the offset is less than or equal to $\frac{N}{2} - 1$ in either direction (left or right). The hit ratio is calculated by dividing the number of cache hits with total third vertex references. These measurements show that the hit rate is 96-98% with just 4-entry cache, which is sufficient to attain high decompression performance.

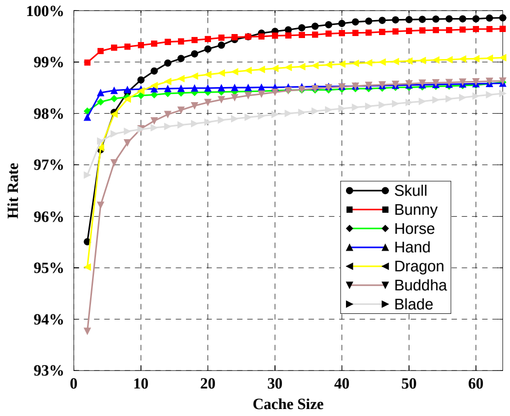


Figure 11. On-chip Cache Hit Rate for BFT.

6 Conclusion

Compression of 3D triangle mesh is a promising approach towards solving the bandwidth problems associated with large 3D meshes. This paper demonstrates the practicability of this approach for high-performance polygonal rendering with prototype FPGA implementation. Experimental evaluation of our approach suggests that compressed 3D mesh representation can significantly reduce the bandwidth requirement in the rendering pipeline, thereby enabling a 3D graphics system to render very large 3D meshes that was not possible with traditional uncompressed approaches.

7 Acknowledgment

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

References

[1] M. M. Chow. Optimized Geometry Compression for Real-Time Rendering. *IEEE Visualization '97*, pages 346–354, November 1997.

[2] M. F. Deering. Geometry Compression. *Proceedings of SIGGRAPH 95*, pages 13–20, August 1995.

[3] F. Evans and E. Azanli. *STRIPE*. `http://www.cs.sunysb.edu/~stripe`, June 1998.

[4] F. Evans, S. S. Skiena, and A. Varshney. Optimizing Triangle Strips for Fast Rendering. *IEEE Visualization '96*, pages 319–326, October 1996.

[5] 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.

[6] S. Gumhold and W. Straßer. Real Time Compression of Triangle Mesh Connectivity. *Proceedings of SIGGRAPH 98*, pages 133–140, July 1998.

[7] H. Hoppe. Optimization of Mesh Locality for Transparent Vertex Caching. *Proceedings of SIGGRAPH 99*, pages 269–276, August 1999.

[8] G. Karypis. *METIS: A Graph Partitioning Software.* `http://www-users.cs.umn.edu/~karypis/metis/metis/index.html`, November 1998.

[9] D. B. Kirk. Unsolved Problems and Opportunities for High-Quality, High-Performance 3D Graphics on a PC Platform. *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 11–13, August 1998.

[10] 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.

[11] 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.

[12] T. Mitra and T. Chiueh. Dynamic 3D Graphics Workload Characterization and the Architectural Implications. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 62–71, November 1999.

[13] J. Rossignac. Edgebreaker: Connectivity Compression for Triangle Meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, January - March 1999.

[14] M. Shand. *PCI Pamette User-Area Interface for Firmware V1.11*. Digital Equipment Corporation, Systems Research Center, February 1999.

[15] G. Taubin and J. Rossignac. Geometric Compression Through Topological Surgery. *ACM Transactions on Graphics*, 17(2):84–115, April 1998.

[16] C. Touma and C. Gotsman. Triangle Mesh Compression. *Graphics Interface '98*, pages 26–34, June 1998.