A New Bandwidth Reduction Method for Distributed Rendering Systems

Won-Jong Lee, Hyung-Rae Kim, Woo-Chan Park, Jung-Woo Kim, Tack-Don Han, and Sung-Bong Yang

Media System Laboratory, Department of Computer Science, Yonsei University, Seoul 120-749 Korea, {airtight, kimhr, chan, mosh, hantack}@kurene.yonsei.ac.kr yang@mythos.yonsei.ac.kr

Abstract. Scalable displays generate large and high resolution images and provide an immersive environment. Recently, scalable displays are built on the networked clusters of PCs, each of which has a fast graphics accelerator, memory, CPU, and storage. However, the distributed rendering on clusters is a network bound work because of limited network bandwidth. In this paper, we present a new algorithm for reducing the network bandwidth and implement it with a conventional distributed rendering system. This paper describes the algorithm called geometry tracking that avoids the redundant geometry transmission by indexing geometry data. The experimental results show that our algorithm reduces the network bandwidth up to 42%.

1 Introduction

As the images become increasingly complex, the size of the model data in 3D graphics grows explosively. A variety of parallel processing techniques have been researched on designing a graphic accelerator to generate high quality images at real time frame rates. Recently scalable displays are highlighted as an important parallel rendering research area. Scalable displays are the graphics hardware/software systems that can generate high resolution(several million pixels) images on the multiple displays.

There are two approaches to build parallel rendering systems for scalable displays. One method is to employ a parallel machine with extremely high-end graphics capabilities. PowerWall[1] and InfinityWall[2] are typical parallel systems. This approach is, however, limited to the number of graphics accelerators that can fit in one computer and is quite expensive. The other method is to utilize the networked clusters of PCs each of which is equipped with a fast graphics accelerator, memory, CPU, and storage.

As compared to high-end parallel computers, there are several advantages of the networked clusters. Since the processors of PCs communicate only by a network protocol, they may be added and removed from the system easily. Because

This work was supported by the NRL-Fund from the Ministry of Science & Technology of Korea.

each PC has its own CPU, memory, AGP bus driving a single graphics accelerator, the aggregated hardware computing, storage, and bandwidth capacity of a PC clusters can grow linearly.

However, a cluster system does not have fast access to the shared virtual memory space and the latencies and bandwidths of inter-processor communication in the system are significantly inferior. Thus the main challenge is to develop efficient parallel rendering algorithms that scale well within the processing, storage, communication characteristics of a PC cluster[3]. AT&T InfoLab[4], Princeton, and Stanford[5] are exploring a variety of parallel rendering techniques for clusters. Stanford developed a software remote rendering system called WireGL[5]. Because it is implemented as a driver that stands in for the system's OpenGL driver, an application can render without modification.

In an immediate-mode graphics API like OpenGL, geometry data are specified by individual function calls. In scenes with significant geometric complexity, an application can perform several millions of such function calls per frame and is limited by the available network bandwidth. Since the network traffic overhead overwhelms the computation overhead on the distributed renderers, it should be a network-bound work rather than a computation-bound work. Thus the network bandwidth reduction scheme is essential.

In this paper, a new method to reduce the network bandwidth for distributed rendering systems is proposed. The patterns of geometry data that dominate network transmission are analyzed. As a result, we have found significant redundancy in the geometry data. The proposed algorithm called *geometry tracking* avoids the retransmission of redundant geometry data by indexing geometry data in a network packet. It is implemented by modifying the WireGL's source code. The experimental results with *SPECViewperf*[6] and *Quake III* are given. They show that our algorithm reduce the network bandwidth up to 42%.

The rest of the paper is organized as follows. Section 2 reviews the WireGL's data transmission scheme and the geometry patterns of the benchmarks are analyzed for measuring its redundant occurrence rate. Section 3 explains the proposed geometry tracking algorithm. Section 4 shows the experimental results. Section 5 concludes the paper.

2 Geometric Redundancy

This section describes the WireGL's data transmission scheme for distributed rendering and analyzes the redundant geometry occurrence rate.

2.1 Data Distribution Scheme in WireGL

WireGL is implemented as a graphics driver that intercepts the application's calls to graphics hardware. WireGL then distributes the application's opcodes and the data to the multiple rendering servers that are responsible for their own tiled displays. Each buffer is placed on the client and distributed servers. The opcodes and the data are stored in each buffer separately. When the geometry buffer is

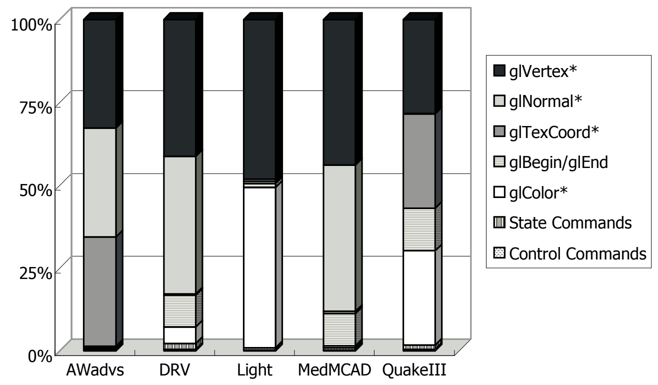


Fig. 1. Comparison of the generated data sizes by the geometry commands and other commands in the OpenGL applications renderings

full or the OpenGL state is changed, the geometry buffer is flushed. At this time the codes and the data in the geometry buffer are packed as a packet which is then transmitted to its corresponding server. That is, the network transmission is performed per packet base, where the size of the packet is the same as that of the geometry buffer.

2.2 Redundancy of the Geometry Data

In order to investigate a method to reduce the network bandwidth, we profile the patterns of the geometry data in all transmitted packets. OpenGL performance benchmarks, SPECViewperf, and the Quake III game are used for the experiment. In the case of the SPECViewperf, the several hundreds of frames of each model are rendered while the demo is rendered in the case of Quake III. Fig. 1 gives the profiling results of the transmitted data to the servers. It shows that the data generated by the geometry commands(`glVertex*, glNormal*, glTexCoord*`) are occupied from 45% to 95% and the size of the data is much larger than that of the data generated by the other commands.

Table 1 provides the average occurring rate of the redundant geometry commands in each packet, whenever the client's geometry buffer is flushed. It shows

Table 1. The average redundant geometry occurrence rate of OpenGL commands in the WireGL network packets

Benchmarks	Number of frames	Number of all vertices	Number of redundant vertices	Rate(%)
AWadvs-04	600	44,013,480	26,000,664	59.07
DRV-07	370	87,499,104	46,790,684	53.48
Light-04	100	62,312,104	38,305,636	61.47
MedMCAD-01	723	1,155,849	403,092	34.87
Quake III	1,346	13,126,642	6,027,431	45.91

that the average redundant occurrence rates of the geometry data in WireGL packets ranges between 35% and 61%. Thus if these redundant data are not retransmitted and the previous transmitted data are reused, the overall network bandwidth will be reduced considerably.

3 Geometry Tracking

In this section, the network bandwidth reduction scheme called geometry tracking algorithm is proposed. Its implementation by modifying WireGL is then described.

3.1 Geometry Tracking Algorithm

The proposed geometry tracking algorithm performs the indexing and the tracking for all geometry data in a single packet. Then, if the same arguments values are occurred repeatedly, the geometry data of the vertex need not be retransmitted. Only its index value is transmitted.

To transmit the index value for the redundant geometry commands and to detect redundant geometry commands in the servers, the following three kinds of new commands are defined. `glIndex_Vertex*, glIndex_Normal*, and glIndex_TexCoord*`. These index commands are used only within the client-servers network and are hidden to applications. The fig. 2, 3 show the geometry tracking algorithm for the client and the server.

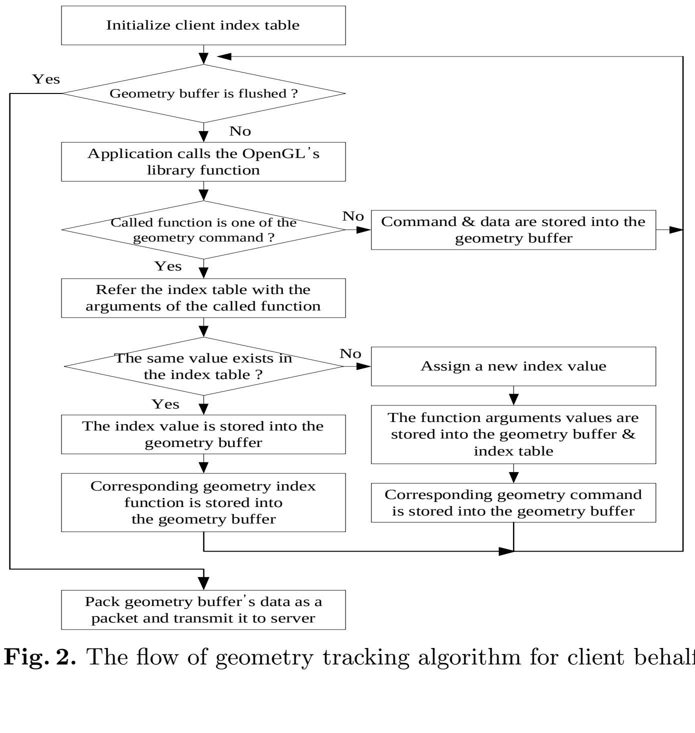

Initialize client index table
Yes
Geometry buffer is flushed ?
No
Application calls the OpenGL's library function
Called function is one of the geometry command ?
No
Command & data are stored into the geometry buffer
Yes
Refer the index table with the arguments of the called function
The same value exists in the index table ?
No
Assign a new index value
Yes
The index value is stored into the geometry buffer
The function arguments values are stored into the geometry buffer & index table
Corresponding geometry index function is stored into the geometry buffer
Corresponding geometry command is stored into the geometry buffer
Pack geometry buffer's data as a packet and transmit it to server

Fig. 2. The flow of geometry tracking algorithm for client behalf

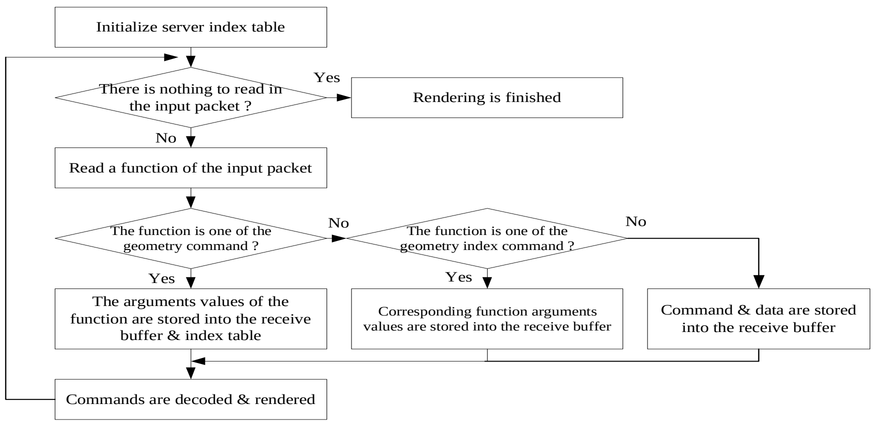

Initialize server index table
There is nothing to read in the input packet ?
Yes
Rendering is finished
No
Read a function of the input packet
The function is one of the geometry command ?
No
The function is one of the geometry index command ?
No
Yes
Yes
The arguments values of the function are stored into the receive buffer & index table
Corresponding function arguments values are stored into the receive buffer
Command & data are stored into the receive buffer
Commands are decoded & rendered

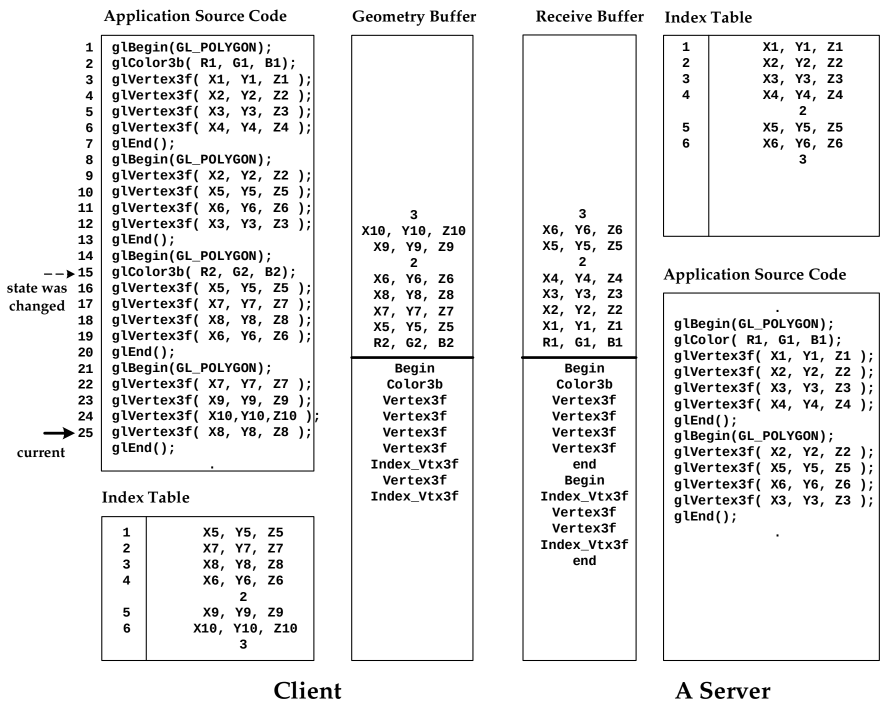
Fig. 3. The flow of geometry tracking algorithm for server behalf

3.2 Implementation on WireGL

On the client's behalf, when the application calls the OpenGL library function, WireGL intercepts application's call to the graphics hardware. If the called function is one of the geometry commands(`glVertex*, glNormal*, glTexCoord*`), the index table is referred with the corresponding function arguments values. If these values already exist in the index table, it means that the same geometry command has been called previously. Thus the corresponding index value is stored into the geometry buffer instead of the arguments values of the called function. The corresponding index command is then encoded and stored. If the arguments values of the called function do not exist in the index table, it means that the same geometry command is not called yet. The arguments values of the called function are stored into the geometry buffer directly and into the index table with the newly assigned index value. This procedure is iterated until the geometry buffer is flushed. Then the geometry data and the index values are transmitted to the distributed servers.

On each server's behalf, the opcodes and the data of the transmitted packet are read in order. If the corresponding function of the opcode is one of the geometry commands, which means that it is not a redundant opcode, thus it is stored into the server's index table and into the receive buffer directly. If the corresponding function of the opcode is one of the index geometry commands(`glIndex_*`), the server's index table is referred with the corresponding index value. Then the arguments values for this function are read from the index table and are stored into the receive buffer. Also the original opcode is restored and is stored into the receive buffer. Finally the opcodes of the receive buffer are decoded and rendered on the corresponding server.

Fig. 4 shows a snapshot after the execution of the 25th line of the application source code. The geometry buffer is flushed either when the buffer is full or when the state is changed. On the client's behalf, since the state of the color was changed in the 15th line as compared with the 2nd line, the geometry buffer was



```
1  glBegin(GL_POLYGON);
2  glColor3b( R1, G1, B1);
3  glVertex3f( X1, Y1, Z1 );
4  glVertex3f( X2, Y2, Z2 );
5  glVertex3f( X3, Y3, Z3 );
6  glVertex3f( X4, Y4, Z4 );
7  glEnd();
8  glBegin(GL_POLYGON);
9  glVertex3f( X2, Y2, Z2 );
10 glVertex3f( X5, Y5, Z5 );
11 glVertex3f( X6, Y6, Z6 );
12 glVertex3f( X3, Y3, Z3 );
13 glEnd();
14 glBegin(GL_POLYGON);
15 glColor3b( R2, G2, B2);
16 glVertex3f( X5, Y5, Z5 );
17 glVertex3f( X7, Y7, Z7 );
18 glVertex3f( X8, Y8, Z8 );
19 glVertex3f( X6, Y6, Z6 );
20 glEnd();
21 glBegin(GL_POLYGON);
22 glVertex3f( X7, Y7, Z7 );
23 glVertex3f( X9, Y9, Z9 );
24 glVertex3f( X10,Y10,Z10 );
25 glVertex3f( X8, Y8, Z8 );
   glEnd();
          .
```

1	X5, Y5, Z5
2	X7, Y7, Z7
3	X8, Y8, Z8
4	X6, Y6, Z6
	2
5	X9, Y9, Z9
6	X10, Y10, Z10
	3

```
      3
X10, Y10, Z10
  X9, Y9, Z9
      2
  X6, Y6, Z6
  X8, Y8, Z8
  X7, Y7, Z7
  X5, Y5, Z5
  R2, G2, B2
--------------
    Begin
   Color3b
   Vertex3f
   Vertex3f
   Vertex3f
   Vertex3f
 Index_Vtx3f
   Vertex3f
 Index_Vtx3f
```

```
      3
  X6, Y6, Z6
  X5, Y5, Z5
      2
  X4, Y4, Z4
  X3, Y3, Z3
  X2, Y2, Z2
  X1, Y1, Z1
  R1, G1, B1
--------------
    Begin
   Color3b
   Vertex3f
   Vertex3f
   Vertex3f
   Vertex3f
     end
    Begin
 Index_Vtx3f
   Vertex3f
   Vertex3f
 Index_Vtx3f
     end
```

1	X1, Y1, Z1
2	X2, Y2, Z2
3	X3, Y3, Z3
4	X4, Y4, Z4
	2
5	X5, Y5, Z5
6	X6, Y6, Z6
	3

```
          .
glBegin(GL_POLYGON);
glColor( R1, G1, B1);
glVertex3f( X1, Y1, Z1 );
glVertex3f( X2, Y2, Z2 );
glVertex3f( X3, Y3, Z3 );
glVertex3f( X4, Y4, Z4 );
glEnd();
glBegin(GL_POLYGON);
glVertex3f( X2, Y2, Z2 );
glVertex3f( X5, Y5, Z5 );
glVertex3f( X6, Y6, Z6 );
glVertex3f( X3, Y3, Z3 );
glEnd();
          .
```

Fig. 4. A snapshot of the proposed geometry tracking scheme

flushed. Then the generated and accumulated data by application source codes between the 1st line and the 15th line were transmitted. On the server's behalf, the geometry data in the receive buffer were read in order and the index table was constructed. The application source codes was then generated by decoding with the opcodes in the receive buffer and the data of the index table.

After the packet was transmitted, the client's index table was reinitialized. In the figure, the opcodes and the data between the 16th line and the 19th line were stored into the index table and the geometry buffer in order. Because the command `glVertex3f` in the 22nd line has the same vertex data as that in the 17th line, the index value of the data in the 17th line, which is 2, was stored into the geometry buffer for the vertex data in the 22nd line. In the same manner, the index value of the vertex data in the 18th line, which is 3, was stored into the geometry buffer for the vertex data in the 25th line.

4 Experimental Results

We built an 8-node cluster that consists of 8 rendering servers as well as a client machine. Each rendering server contains a Pentium IV 1.6GHz processor and an NVIDIA GeForce3 Ti 200 graphic accelerator. The client machine contains a dual AMD Athlon XP 1800+ 1.5GHz and an NVIDIA GeForce3 Ti 200 graphic accelerator. The cluster is connected with a high speed network. Each rendering

Fig. 5. Three benchmarks to be experimented. DRV-07, Atlantis, Quake III

server outputs a 1024x768 resolution video signal. We experimented on this cluster with the implemented software. Fig. 5 shows the benchmarks to be tested our system for the experiment.

1. *DRV-07* is a product of the SPECViewperf. It contains 367178 vertices in 42821 primitives and its size is greater than 50 megabytes. It is used as a benchmark because of its high scene complexity compared with other SPECViewperf products.
2. *Atlantis* is one of the OpenGL demo programs. It simulates a pool of swimming sharks, whales, and a dolphin. The body poses for each object are computed in real time. Because it is rendered infinitely, it is limited by 500 frames.
3. *Quake III* is one of the typical current video games and is frequently used as a benchmark. One of its demos that contain 280 frames is tested.

The number of vertices in a packet can be increased since the redundant vertex can be stored with the small size into the geometry buffer by the geometry tracking method. In Table, the geometry tracking method and WireGL are compared in terms of the number of overall transmitted packets. They are tested the single rendering server and client. As shown in Table 2, it can be reduced up to 45%.

Each application is tested on four different server configurations: 1x1, 1x2, 2x2, and 2x4 until each rendering is finished. Fig. 6 shows that the total sizes of the transmitted data as the number of servers increases. In order to compare with WireGL's data sent to the servers, the relative transmission rate which is the ratio of the data sent to the servers to WireGL's data sent to a 1x1 configuration is measured. According to Fig. 6, the traffic reduction rate in DRV-07 ranges between 11% and 27%, the rate in Atlantis ranges between 34% and 42%, and the rate in Quake III ranges between 8% and 12%. Since Quake III has heavy texture images that should be sent to each sever, the reduction rate is not better

Table 2. Comparison of the number of overall transmitted packets

	DRV-07	Atlantis	Quake III
WireGL	774,806	2,723	10,228
Geometry tracking	494,572	1,489	9,096
Reduction rate (%)	36.17	45.32	11.08

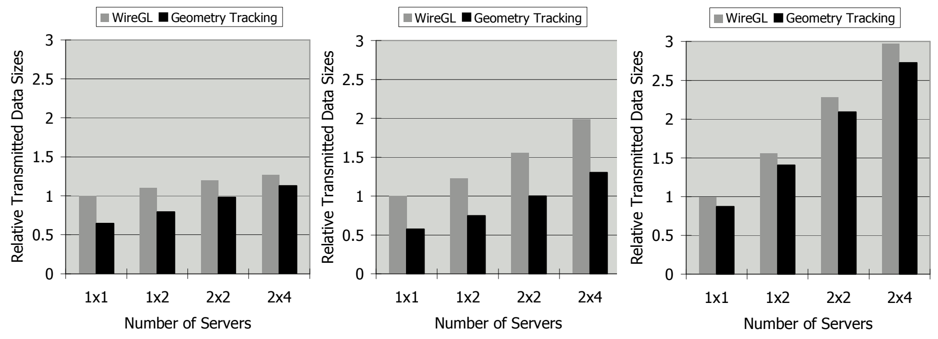


Fig. 6. Relative transmitted data sizes with increasing number of servers. DRV-07, Atlantis, Quake III

than those of other benchmarks. Consequently geometry tracking can reduce the network bandwidth up to 42%.

5 Conclusion

This paper has introduced a new algorithm called geometry tracking that can reduce the network bandwidth by indexing the geometry data. The geometry tracking algorithm can reduce the network bandwidth up to 42%. However, the texture-intensive application like a Quake III cannot reduce the rate as much as other applications can because of its heavy texture traffic.

The geometry tracking algorithm will be extended to reduce the texture traffic and will be used for other applications that have more redundancy such as volume rendering. The algorithm is expected to improve the overall rendering performance for volume rendering systems.

References

1. University of Minesota PowerWall, `http://www.lcse.umn.edu/research/powerwall/powerwall.html`
2. Marek, C., Dave, P., Daniel, S., Tom, D., Gregory, L.D., Maxine, D.B.: The ImmersaDesk and InfinityWall Projection-based Virtual Reality Displays, ACM SIGGRAPH Computer Graphics, Vol. 31. ACM Press, New York (1997) 46-49
3. Dirk, B., Bengt-Olaf, S., Claudio, S.: Rendering and Visualization in Parallel Environments, Proceedings of the SIGGRAPH 2000 Course Notes (2000)
4. Bin, W., Claudio, S., Eleftherios K., Shankar K., Stephen N.: Visualization Research with Large Displays, IEEE Computer Graphics and Applications, Vol. 20, No. 4, July/Aug. (2000) 50-54
5. Greg, H., Matthew, E., Ian, B., Gordan, S., Matthew, E., Pat, H.: WireGL: A Scalable Graphics System for Clusters, Proceedings of the 2001 Conference on Computer Graphics (2001) 129-140
6. SPECViewperf, `http://www.spec.org/gpc/opc.static/opcview.htm`